Efficient Analog Circuit Modeling By Boolean Logic Operations

Zhenyu Qi, and Sheldon X.-D. Tan, Pu Liu

Department of Electrical Engineering
University of California, Riverside, CA 92521, USA
{ zhenyu, stan, uliu}@ee.ucr.edu

ABSTRACT

Abstract— *In this paper, we propose a novel symbolic analysis method for analog behavioral modeling by Boolean logic operations and graph representation. The exact symbolic analysis problem is formulated as a logic circuit synthesis problem where we build a logic circuit which detects whether or not a given symbolic term is a valid product term from a determinant. The logic circuit is represented by binary decision diagrams (BDDs), which can be trivially transformed into zero-suppressed binary decision diagrams(ZBDDs). ZBDDs are essentially determinant decision diagrams (DDDs) representation of a determinant. The significance of the new method is that all product terms can be constructed implicitly and simultaneously, in contrast to all previous symbolic analysis methods where symbolic terms are generated explicitly and sequentially by Laplace expansion or topological methods. We further apply the logic synthesis idea to generating symbolic coefficients of s-expanded polynomials and present a method to compute coefficients individually and selectively. Our new approach demonstrates an inherent relationship between circuit simulation and logic synthesis for the first time. Experimental results show the speedup of our new method over the existing flat method and its greater capacity over both existing flat and hierarchical symbolic analyzers.*

1. INTRODUCTION

Symbolic analysis is to calculate the behavior or the characteristics of a circuit in terms of symbolic parameters. It is a viable approach to deriving parametric behavioral models of analog and other time-continuous modules. As illustrated in [2], simple yet accurate symbolic expressions can also be interpretable by analog designers to gain insight into circuit behavior, performance and stability, and are important for verification and synthesis applications in analog circuit design automation [3].

Exact symbolic analysis of a linear system, however, is a hard problem due to the exponential growth of symbolic expressions with the size of circuits [2]. To alleviate this long-standing problem, a graph-based approach was proposed in [8] that uses a special graph called determinant decision diagrams (DDDs) to represent the generated product terms from the expansion of a determinant. DDD representation has led to exponential reduction of symbolic expressions because the nodes of a DDD graph grow much more slowly than the DDD graph paths, which represent symbolic product terms [8].

Hierarchical approach is another effective way to cope with large analog circuits. Hierarchical decomposition is to generate symbolic expressions in the *sequence-of-expression* forms [5, 10, 12]. There are three methods known as topological analysis [10], network formulation [5] and DDD-based approach [12]. The major drawback for all those hierarchical based exact symbolic analysis is the generated sequence of expressions is difficult to interpret and manipulate. Recently a hierarchical DDD construction algorithm was proposed in [11] that overcomes the previous hierarchical analysis problems and extends the exact symbolic analysis capacities.

One common problem with existing symbolic analysis is that the symbolic terms have to be generated explicitly and sequentially by Laplace expansion [4] or topological methods [6], which may lead to exponential construction time even the final DDD sizes do not grow exponentially. This is an important reason that symbolic analysis has suffered from the long-standing circuit size problem. The introduction of DDD graph only solves the representation side of this problem.

In this paper, we look at the generation side of the symbolic analysis problem. We propose a novel approach to generating all the symbolic expressions *implicitly* and *simultaneously*. Our approach is inspired by the recent symbolic approach to pointer analysis for compilation optimization [13] where logic functions are used to construct the symbolic invocation graphs. The main idea of the new approach is that the symbolic expression generation is viewed as a logic circuit synthesis process, and we design a logic circuit that can detect whether or not a symbolic term a valid product term from a determinant. The logic circuit, which is essentially a Boolean function, can be represented by binary decision diagrams (BDDs). BDDs are then trivially transformed into zero-suppressed binary decision diagrams(ZBDDs), which are essentially DDD representation of the determinant.

Along the same line, we can design a dedicated coefficient conditional logic and use the same BDD-based logic synthesis idea to construct symbolic coefficients of the *s*-expanded polynomial of a determinant. Unlike previous methods, the new algorithm allows constructing individual coefficients directly, and is able to derive symbolic *s*-polynomials for very large analog circuits.

The most important advantage of the new approach over existing ones is that the time complexity is no longer tied to the number of product terms but depends on the implicit representation of designed logic during the entire construction process. This makes the symbolic analysis problem much more tractable as sizes of BDD/DDD graphs typically grow very slowly with circuit sizes given a good variable ordering. The new symbolic analysis method shows an inherent relationship between circuit simulation and logic synthesis for the first time.

The rest of the paper is organized as follows: Section 2 briefly reviews concepts of binary decision diagrams and determinant decision diagrams as both are used in the new method. Section 3 presents the logic circuits for detecting valid product terms from a determinant. Sections 4 shows an improved approach to constructing DDDs by logic operations directly from a matrix based on the logic circuit. Section 5 introduces the logic operation based *s*-expanded DDD method. Section 6 gives a brief time complexity analysis of the given method. Section 7 presents experimen-

*This work is funded by NSF CAREER Award CCF-0448534, NSF Grant OISE-0451688 and UC Regent's Faculty Fellowship(04-05).

tal results and the comparison with existing DDD-based methods. Section 8 concludes the paper.

2. BINARY DECISION DIAGRAMS AND DETERMINANT DECISION DIAGRAMS

In this section, we briefly review the concepts of binary decision diagrams for representing Boolean functions and determinant decision diagrams for representing determinants.

A binary decision diagram (BDD) is an ordered, directed graph representation of a Boolean function, as shown in Fig.1(a), which represents the Boolean function $f = x_3x_2 + \overline{x}_1x_2$. A BDD has two terminal vertices, namely the 0-terminal vertex and the 1-terminal vertex. Each non-terminal vertex has two edges, called 0-edge and 1-edge. Each path from the root vertex to the 1-terminal represents a product in the Boolean function expressed in the sum-of-product form. Such BDD is also called reduced ordered BDD (ROBDD) as it is obtained by eliminating all the redundant nodes whose two edges point to the same node in the binary tree graph [1]. If the variable ordering is fixed, BDD gives a canonical representation of a Boolean function [1].

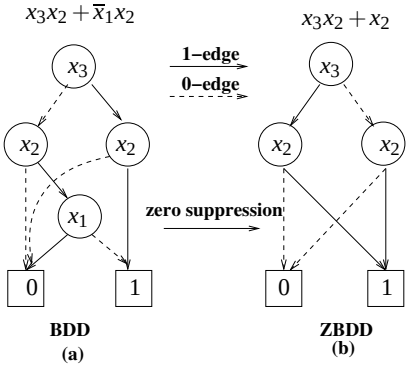

Figure 1: BDD versus ZBDD.

A determinant decision diagram is a signed zero-suppressed BDD (ZBDD) representing the determinant of a matrix [8]. ZBDD was introduced for representing combination sets [7]. The corresponding ZBDD of the BDD in Fig. 1(a) is obtained by performing the zero suppression rule. The resulting ZBDD actually represents combinational set $\{x_3x_2, x_2\}$. The suppressed vertices, which have their 1-edge pointing to the 0-terminal, essentially correspond to all the negatively valued Boolean variables (like $\overline{x}_1$) in Fig. 1(a). Note that a BDD can be trivially transformed into a ZBDD, while a ZBDD can't be directly transformed to a BDD without a well defined universal set. If we use ZBDD to represent all the product terms in a matrix determinant where each product term is treated as a combination of entries in the matrix, and associate each vertex with a unique sign, the ZBDD graph becomes the DDD graph [8].

3. BOOLEAN LOGIC FOR DETECTING THE TERMS IN A DETERMINANT

The DDD graph is introduced to represent a determinant. It essentially represents all the product terms in the determinant. In a DDD graph, each product term corresponds to an 1-path from the root vertex to the 1-terminal. If we view a DDD graph as a BDD graph, where each symbol in a product term takes *true* Boolean value, all the other symbols take *false* Boolean value, then the DDD essentially represents the logic that detects if a given symbolic term is a product term in the determinant, as a valid product term always corresponds to an 1-path, and thus satisfies the logic.

This motivates us to generate the DDD graph by constructing a logic circuit which is able to detect if a given product term is a valid one from the determinant. This turns out to be an easy design problem. Indeed, from the definition of determinant [4], we can design a logic to check whether the rows and columns of all the elements in a symbolic term cover every row and column of the matrix exactly once.

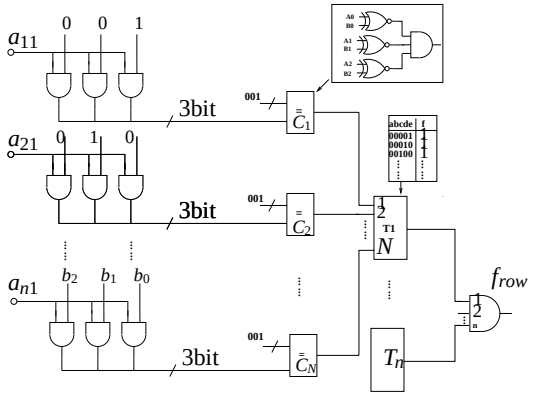

Figure 2: The logic circuit for detecting a valid product term from a determinant.

Fig. 2 shows a portion of the logic schematic for checking whether a given product term is valid from an $n \times n$ matrix. We simply compare the row/column index of each nonzero element in this product term with the index of each row/column and examine if each row/column index appears exactly once.

The logic in Fig. 2 checks for row 1 (encoded as 001 since 3 bit binary coding is used in this example). $a_{11}, a_{21}...a_{nn}$ are the elements in the product term to be checked, $001, 010..., b_2b_1b_0$ are the binary codes for all row indices in the matrix. T_1 is true only when one of its inputs is true, ensuring that exactly one nonzero element is in row 1. Comparators C_1 to C_N compare the row index of each nonzero element with the row index of row 1. (N is the total number of nonzero elements in the matrix). The *AND* gate in the last stage makes sure that all the row indices of the matrix are present in the product term. The resulted Boolean function for the row index legality check is f_{row}.

We can do the same for the column index legality check where each nonzero element is compared with the column index of each column. The resulting logic function for column index legality check is f_{col}. Since both row and column legality conditions must be satisfied to make a valid product term, the final logic is the *conjuncture (AND operation)* of two logic functions:

$$f_{det} = f_{row} \wedge f_{col} = f_{row} f_{col} \tag{1}$$

where $\wedge$ operation is the logic AND operation. We may also write the $f_{row} \wedge f_{col}$ as $f_{row} f_{col}$ in the sequel. The resulting logic f_{det} is the Boolean logic we are looking for.

4. NEW LOGIC OPERATION BASED DDD CONSTRUCTION ALGORITHM

In this section, we show that the logic circuit shown in Fig. 2 can be further simplified and the DDD construction can be performed efficiently by a number of simple logic operations.

4.1 Efficient BDD Construction For the Determinant Detecting Logic

For the determinant detecting logic circuit in Fig. 2, we observe that if the nonzero element a_{ij} is not in row 1, then the comparison result will always be 0 (i.e. C_i is always 0). On the other hand, if the a_{ij} is in row 1, the C_i will be a_{ij} where a_{ij} is a Boolean variable. Suppose that row 1 has three nonzero elements a_{11} a_{12} and a_{13}, then we have

$$T_1 = a_{11}\overline{a}_{12}\overline{a}_{13} + \overline{a}_{11}a_{12}\overline{a}_{13} + \overline{a}_{11}\overline{a}_{12}a_{13}, \tag{2}$$

where "+" is the OR operation. As a result, we conclude that each nonzero element in a row i will generate a product term for each row's uniqueness checking function T_i. In the product term of each nonzero element, the corresponding nonzero element will take *true* Boolean value while the rest nonzero elements in the same row will take false Boolean value. So every nonzero element in a determinant will generate one product term for constructing f_{row}.

For a $n \times n$ matrix, the row legality checking function f_{row} become:

$$f_{row} = T_1 \wedge T_2 ... \wedge T_n \tag{3}$$

We do the same for generating the column legality check function f_{col} where every nonzero element generates one product term also for f_{col}. We can directly build those product terms from a determinant by inspection, which simplifies the BDD construction considerably. Theoretically, we have

THEOREM 1. *A product term is a valid one product term of a given matrix determinant $det(A)$ if and only if (after the product term is transformed into a Boolean expression), it satisfies the Boolean function $f_{det(A)}(= f_{row} \wedge f_{col})$. f_{row} and f_{col} are defined above for determinant $det(A)$.*

Proof: First, we look at the 'if' part of the theorem. Let's look at a product term with all possible combinations of matrix elements. First, if it does not include any element from a certain row or column j, then all variables in the corresponding T_j are evaluated to zero, which makes $T_j = 0$, which in turn makes $f_{det(A)} = 0$ since all T_i's are ANDed together. Second, if it has more than one element from a certain row or column j. This again makes $T_j = 0$ and immediately makes $f_{det(A)} = 0$. So the only possibility a product term satisfies $f_{det(A)}$ is when it has exactly one element from each row and each column, which is the exact requirement for a valid product term from the matrix determinant $det(A)$.

Then we prove the 'only if' part of the theory. Based on the definition of matrix determinants, a product term in a determinant has one and only one element from each row and each column. The legality requirement is precisely expressed in Boolean functions in T_i, $i = 1, 2...n$. To enforce the requirement that every row and column of a product term exists only once, we need to AND all those T_i's from all the rows and columns, which actually is the Boolean expression $f_{det(A)} = f_{row} \wedge f_{col}$.

In the following, we illustrate such construction using a simple 2×2 determinant $det(A_{2\times 2})$ as shown below:

$$det(A_{2\times 2}) = \begin{vmatrix} a_{11} & 0 \\ a_{21} & a_{22} \end{vmatrix} = a_{11}a_{22}.$$

Determinant $det(A_{2\times 2})$ only has one product term $a_{11}a_{22}$. We now show how this product term can be generated by using the aforementioned logic circuit.

First, we construct row legality check Boolean function f_{row}. For row 1, we have $T_{r,1} = a_{11}$. For row 2, we have $T_{r,2} = a_{21}\overline{a}_{22} + \overline{a}_{21}a_{22}$. As a result, f_{row} becomes

$$f_{row} = T_{r,1} \wedge T_{r,2} = a_{11}(a_{21}\overline{a}_{22} + \overline{a}_{21}a_{22})$$

Then we construct column legality check Boolean function f_{col}. For column 1, we have $T_{c,1} = a_{11}\overline{a_{21}} + \overline{a}_{11}a_{21}$. For the column 2, we have $T_{c,2} = a_{22}$. As a result, f_{col} becomes

$$f_{col} = T_{c,1} \wedge T_{c,2} = a_{22}(a_{11}\overline{a_{21}} + \overline{a}_{11}a_{21})$$

The final BDD representing all the product terms from $det(A_{2\times 2})$ is

$$\begin{aligned} f_{det(A_{2\times 2})} &= f_{row} \wedge f_{col} \\ &= (a_{11}(a_{21}\overline{a}_{22} + \overline{a}_{21}a_{22}))(a_{22}(a_{11}\overline{a_{21}} + \overline{a}_{11}a_{21})) \\ &= a_{11}a_{22}\overline{a}_{21}. \end{aligned}$$

Boolean expression $a_{11}a_{22}\overline{a}_{21}$ actually is exactly the BDD representation of the valid product term $a_{11}a_{22}$ as $\overline{a}_{21}$ will be suppressed when the BDD graph is transformed into ZBDD graph (DDD). Note that the sign of each node in the DDD will be computed when the DDD is constructed from the corresponding BDD.

4.2 New Construction Algorithm

In this subsection, we outline the new BDD construction algorithm for determinant detecting logic shown in Fig. 2. For a nonzero element a_{ij} at row i, let $P_r(a_{ik})$ designate the product term where a_{ik} takes *true* Boolean value while the rest nonzero elements in row i take *false* Boolean value, $\overline{a}_{il}, l \neq k$. The same is true for product term $P_c(a_{jk})$ for a nonzero element a_{kj} in a column j. Then the BDD construction algorithm is given in Fig.3.

```
BDDCONSTRUCTBYLOGIC (A) {
For each row i in matrix A
   T_{r,i} = Σ_{k=1}^{n} P_r(a_{ik})
   f_row = f_row ∧ T_{r,i};
For each column j in matrix A
   T_{c,j} = Σ_{k=1}^{n} P_c(a_{jk});
   f_col = f_col ∧ T_{c,j};
f_{det(A)} = f_row ∧ f_col;
return f_{det(A)};
}
```

Figure 3: BDD construction algorithm for the determinant detecting logic.

It can be seen that BDD construction boils down to a number of *AND* operations. We just *AND* all $T_{x,i}$ from every row and column. Once the BDD is constructed, DDD is obtained by suppressing all the vertices with their 1-edge pointing the 0-terminal. This can be done trivially by one traversal of the BDD graph.

4.3 Logic Synthesis Perspective

Although the DDD construction process can be simplified into a sequence of simple logic operations, we stress that the main idea of the new method is still based on the logic synthesis concept: we generate the desired symbolic expression in terms of DDD graphs (for a determinant, its cofactor or the coefficients of its s-polynomials) by constructing proper logic circuits. So we need to first design the circuits as shown in Fig. 2 and Fig. 4. Once those logic circuits are designed, we can represent such circuits in terms of BDDs. In this section, we mainly show that such a transformation process can be further simplified into a number of simple Boolean operations for the construction of DDDs. Actually for the construction of the s-expanded coefficient DDD by logic synthesis based methods, we do not have the simplified logic operations.

5. *s*-EXPANDED POLYNOMIAL CONSTRUCTION BY LOGIC OPERATIONS

In this section, we show how to construct the symbolic coefficients of the *s*-polynomial of a determinant using Boolean logic operation method.

5.1 Review of Existing DDD-based Construction Method

An *s*-expanded polynomial of a determinant takes the form of

$$P(s) = a_0 s^0 + a_2 s^1 + ... + a_n s^n. \tag{4}$$

In [9], a DDD graph like those in previous sections (which is referred to as *complex DDD*) of a circuit matrix or its cofactors is built first. Then the s-expanded polynomial in terms of multi-root DDDs is constructed from the complex DDD. In this method, coefficient DDDs of all orders in the polynomial need to be generated, which is expensive, as the highest order is the size of the matrix. More importantly, this is often unnecessary, as dominant poles are usually associated with low-order coefficients of the denominator polynomial.

5.2 Generation of the Coefficient of a Specific Order of s

With the new logic synthesis idea, we can construct each individual coefficient s-expanded DDD one at a time. To illustrate this, we use the 2×2 example in Section 4.1 again with the assumption that it can be further expanded as below:

$$\begin{vmatrix} a_{11} & 0 \\ a_{21} & a_{22} \end{vmatrix} = \begin{vmatrix} a+bs & 0 \\ c+ds & e+fs \end{vmatrix} = ae + (af+be)s + bfs^2$$

where $a_{11} = a + bs$, a is a resistive admittance and b can be a capacitive or inductive admittance. a_{21} and a_{22} are similar. Suppose we are interested in symbolic coefficient of s^1, which is $(af+be)s$.

The basic idea is to construct a logic which filters out incorrect product terms and retains correct combinations. The desired logic is the conjuncture of the following two conditions: first, the product term should belong to the determinant; second, it has the right order of s. In this case, it contains only one s (order one). For instance, afs and bes are all the legitimate product terms, whereas product terms like ads fails the first condition and bfs^2 fails the second condition.

The first logic, which we still name f_{det}, is essentially the same as the one constructed in Section 4.2. The only difference is that the row and column legality check in Eq.(3) are now performed on each individual admittance instead of nonzero entries in the matrix, since no two admittances from the same row or column should appear concurrently in any valid product term. For example, the $T_{r,2}$ now becomes

$$T_{r,2} = c\,\overline{ds}\,\overline{e}\,\overline{fs} + \overline{c}\,ds\,\overline{e}\,\overline{fs} + \overline{c}\,\overline{ds}\,e\,\overline{fs} + \overline{c}\,\overline{ds}\,\overline{e}\,fs$$

So the number of products in each T_x is the number of admittances in each row or column.

The second logic circuit is to find the all the product terms with correct order of s. We call such logic circuit *coefficient condition logic* f_{order}. Let's assume first that all n admittances are reactive (capacitive and inductive).

The corresponding logic circuit is shown in Fig. 4, which basically just counts the number of reactive elements (with complex variable s) and then compares the number with a fixed number (desired order). In other words, this logic circuit essentially detects if a given product term has the desired order (given as m) of s. f_{coeff} is the resulting coefficient BDD of a certain required order. f_{det} is the output of logic to detect if a given product is valid product term or not of a determinant as shown in Fig.2.

In this logic circuit, we have n k-bit full adders cascaded together. One k-bit input of the full adders is always zero and the other k-bit input is driven by the output of the previous stage adder. In each adder stage, the capacitive or inductive admittance variable y_i drives the carry-in c_0. The basic idea is to count the number of those reactive admittances in a product terms. So the sum, which is the output of the last stage adder will be compared with a fixed binary number. If the fixed number is m, the logic circuit will be satisfied when the number reactive admittance in the given product term is m.

Then we have the following theorem:

THEOREM 2. *Given a set of n Boolean variables* $\{y_1 y_2 ... y_n\}$, *the coefficient condition logic* f_{order} *in the circuit of Fig. 4 is*

$$f_{order} = y_1 y_2 ... y_m \overline{y_{m+1}} ... \overline{y_n} + y_1 y_2 ... \overline{y_m} y_{m+1} \overline{y_{m+2}} ... \overline{y_n} + ... + \overline{y_1 y_2} ... \overline{y_{n-m}} y_{n-m+1} ... y_n \,, \quad (5)$$

where f_{order} *is sum of* $C_m^n (= \frac{n!}{m!(n-m)!})$ *product terms and each term consists of n Boolean variables, among them, m take true value and the rest* $(n-m)$ *variables take false value.*

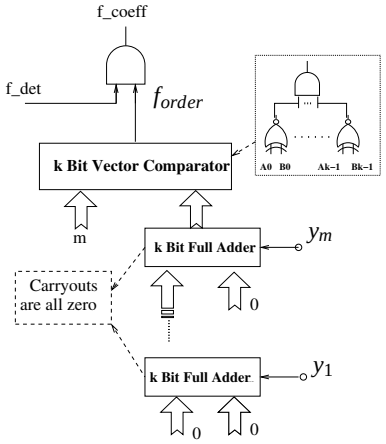

Figure 4: The logic circuit for coefficient generation in Theory 2

Proof: Logic expression f_{order} defined in Eq.(5) is satisfied *if and only if* there are m variables among all the inputs $y_i, i = 1...n$ are logic *true*, *and* the rest $n-m$ variables are logic *false*, as one of the product terms in Eq.(5) will be evaluated to '1'. On the other hand, the logic circuit in Fig. 4 will be evaluated to '1' when there are m y_i are '1' assuming that k is larger enough so that there is no overflow for all addition operations. Therefore, the logic circuit for f_{order} in Fig. 4 corresponds to the logic expression of Eq.(5).

With Theorem 2, the second logic which selects the right order number can be easily constructed. We simply take all the reactive admittances with s (capacitors and inductors) as input Boolean variables, and choose m to be the order number desired. This logic is exactly what we are looking for. All the adders in Fig. 4 are of bit k, which is large enough so that the there is no overflow for all adders. In practice k is the 2-based logarithm of the highest possible s-order of the network function, or the circuit matrix size.

So the final logic for coefficient construction is obtained by $f_{coeff} = f_{det} \wedge f_{order}$, as represented in Fig. 4. Notice that admittances without s (like those related to resistors) would not contribute in f_{order}. Say, if the s^0 term is desired, we just choose $m = 0$ and the logic is simply $\overline{y_1 y_2} ... \overline{y_n}$, which means any admittance with s should be discarded from the s^0 term.

5.3 Generation of Terms with Several Specific s-Orders

The logic introduced in 5.2 only generates terms with one specific s-order. As pointed out in 5.1, the first few s-order terms are often of particular interest. For this purpose, we can certainly repeat the above operation for several times, which is in fact not so efficient. Notice that in Fig. 4, the conditional logic before the final comparator remains the same for all coefficients of different orders of s. Once the logic is generated for a certain order m, the only change required for another interested order l is to replace one of the comparator inputs from m to l. The rest of the logics in Fig. 4, as well as the logic for the determinant generation, remain the same.

5.4 Cancellation Removal

The symbolic cancellation can be easily handled in the proposed method. If term ab cancels with cd symbolically, we just multiply (AND operation) the generated coefficient BDD, f_{coeff}, with product terms $a\overline{b}$, $\overline{a}b$, $\overline{a}\overline{b}$, $c\overline{d}$, $\overline{c}d$, $\overline{c}\overline{d}$. In other word, we do not allow product terms with both a and b, c and d appearing at the same time. This is exactly the cancellation removal process.

6. TIME COMPLEXITY ANALYSIS

The time complexity of the proposed method can roughly be related the general time complexity of BDD operations, which are proportional to sizes of the resulting BDD graphs of two operations. But the sizes of the BDD graph are highly depends on the variable ordering, which in the best case has linear time complexity and in the worst case (parity functions) will still have exponential growth with size of the number of Boolean variables (circuit sizes in our case). But many practical circuits have very small BDD sizes compared to the number of their minimum product terms, which makes BDD methods very useful for many logic synthesis and verification applications. In our BDD/DDD based symbolic analysis, we see the similar time complexity. But from symbolic analysis perspective, such time complexity is significant as the time complexity is no longer related to the number of product terms any more. Instead it depends on the size of BDDs representing the product terms at all the time.

7. EXPERIMENTAL RESULTS

The proposed algorithm has been implemented. A number of analog circuits ranging from large Opamp circuits, active filters to mesh-structured RC filter circuits are analyzed symbolically. All results are collected on a Linux workstation with dual 3.0Ghz Xeon CPUs and 2GB memory.

We first test the new algorithm on a number of full matrices with different sizes. The results are also compared with the Laplace based DDD construction algorithm as shown in Fig. 5. Notice that the total number of product terms of a full $n \times n$ determinant is $n!$. We found that Laplace expansion can only construct DDDs for up to 11×11 full matrix in reasonable time (less $<$ 10hrs) and memory (less $<$ 1GB), while logic operation based algorithm can construct DDDs for up to 16×16 full matrix. Notice that the difference of the number of product terms between a full 16×16 determinant and a full 11×11 one is about six orders of magnitude ($2.09 \times 10^{13} - 2.99 \times 10^{7}$).

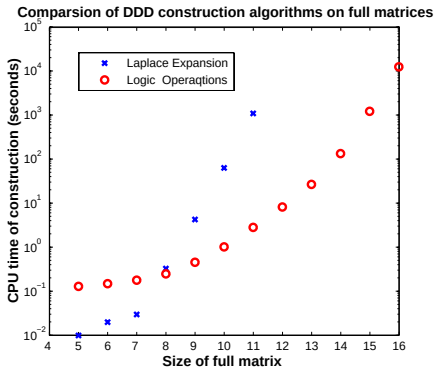

Figure 5: CPU time comparison of Laplace expansion and logic operation DDD construction algorithms.

Next, we test the new program on a number of analog circuits and structured RC filter circuits. The selected circuits are three Opamp circuits, *CMOS_Opamp*, $\mu A741$, $\mu A725$ [11], one active band-pass filter [10], and a number of mesh-structured RC circuits. For instance, circuit $p5x20x2$ has 5 rows and each row is an RC ladder circuit with 20 RC segments. The last digit 2 is the number of the vertical lines for connecting the 5 RC ladder chain circuits at evenly distributed points. Circuit $p10x100$ or similar other circuits are tree-like structured circuits.

First we compared our new determinant construction algorithm with the Laplace expansion algorithm and the exact hierarchical DDD algorithm [11], which is the most efficient exact symbolic analysis algorithm reported so far. We compared the resulting sizes of DDD graphs for the denominator of a transfer function for each circuit and CPU time of the DDD construction for the whole transfer functions. Then we generate s-expanded rational functions using logic operation based method and compare with method in [9], which is the most efficient fully s-expanded polynomial construction method reported.

Table 1 summarizes the results. In this table, *#nodes* is the number of nodes in the circuits and *#paths* is the number of product terms in the denominator of the transfer function. $|DDD|(LS)$ and $|DDD|(H)$ are the number of DDD nodes for denominators of a transfer function for logic operation based and hierarchical methods respectively. Notice that Laplace expansion method uses the same circuit matrix and variable ordering as that of logic operation based method. So their DDD size should be the same. $CPU(F)$ gives the CPU time using Laplace expansion method to construct the required transfer functions; $CPU(H)$ and $CPU(LS)$ are the CPU times for the hierarchical construction method and logic operation based method respectively.

The results for constructing coefficients of s-expanded polynomials are listed in the last two columns, where $s\text{-}exp(F)$ refers to method in [9] and $s\text{-}exp2(LS)$ refers to the proposed logic operation-based approach. Assume that we are only interested in the lowest 7 orders of both coefficients in both the numerator and denominator of the transfer function.

From Table 1, we observe first that the new algorithm can construct the DDD graphs for all the cases, while hierarchical method fails for some cases. The failure typically comes from excessive memory use as approximately only 1GB is available for user programs in our Linux computer. For the hierarchical method, we typically use 5 or 10 as the limit for the number of internal nodes for each subcircuit as they are the best subcircuit sizes for exact hierarchical analysis as shown in [11]. For Laplace expansion based method, only very small circuits can be analyzed and the new method has much greater capacity.

We observe also that in addition to circuit size, variable ordering plays a critical role in construction time. However, the hierarchical and logic operation based algorithms do not use exactly the same variable ordering as variable ordering is computed for each subcircuit in the hierarchical method. As a result, there is no consistent speedup trend between these two methods. For some smaller circuits like *band_pass*, the hierarchical method fails to deliver results because the variable ordering used make the construction process very slow and memory intensive, which eventually runs out of memory.

But in general, we find that logic operation based construction method will lead to smaller DDD sizes than the hierarchical method. The reason is that variable ordering for the new method is done in the entire circuit, while the variable ordering is done for each subcircuit separately in the hierarchical method. But there is an exception: for circuit $\mu A741$, DDD sizes are 2314 for hierarchical method versus 4985 for the new method. This suggests DDD variable ordering algorithm, which is mainly based on the Markorwtiz's algorithm [8] has much room for further improvement for many practical and unstructured analog circuits. For mesh-structured circuits, the hierarchical method seems faster than the new algorithm although they use more DDDs to represent the final expressions. On the other hand, when the structures become more complicated (more vertical lines) as in the cases of *p6x20x4* and *p7x20x4*, hierarchical method fails to give any result.

Table 1: DDD Construction results on the different analog circuits.

Circuit	#nodes	#paths	$\|DDD\|$(LS)	$\|DDD\|$(H)	CPU(F)	CPU(H)	CPU (LS)	s-exp(F)	s-exp(LS)
CMOS_Opamp	14	79643	1895	8803	0.44	2.21	0.11	12	7
μA741	24	108032	4985	2314	0.60	0.25	0.30	425	74
band_pass	38	7.95×10^8	39056	–	–	–	3.31	-	743
p5x20x2	100	5.53×10^{20}	3312	18631	–	2.90	1.30	-	22
p5x20x4	100	2.10×10^{21}	18791	392326	–	202.63	3.01	-	124
p6x20x2	120	9.17×10^{24}	6800	18701	–	2.87	4.90	-	85
p6x20x4	120	4.83×10^{25}	41599	–	–	–	10.31	-	3764
p7x20x2	140	1.52×10^{29}	13776	25263	–	5.61	5.61	-	341
p7x20x4	140	1.11×10^{30}	87215	–	–	–	22.31	-	10620
p10x100	1000	N/A	5647	67840	–	17.02	44.06	-	-
p20x100	2000	N/A	11597	150400	–	92.59	196.34	-	-
p10x300	3000	N/A	17047	–	–	–	1447.10	-	-
p20x200	4000	N/A	23297	–	–	–	2651.77	-	-

We also notice that CPU time for p5x20x4 is significantly larger compared with p5x20x2. The reason is that circuit p5x20x4 has more complicated circuit structure than p5x20x2, as p5x20x4 has 4 vertical RC lines compared with 2 vertical RC lines in p5x20x2. We observe that circuit structure has significant impacts on the CPU time of BDD/DDD operations with our simple variable ordering algorithm.

For tree-like structured circuits, we find that the number of paths is out of numerical range of floating point number in the computer as indicated by N/A. For circuits $p10x300$ and $p20x200$, the hierarchical method also fails to construct DDDs due to limited memory resource. One critical engineering problem with BDD or DDD is that we need to do garbage collection (CG) to recycle unused DDD nodes. This is especially important for logic operation intensive applications like the hierarchical and the new method as many BDD/DDD logic "AND" operations are used. The garbage collection can be done very easily in the new method as we perform the GC at the end of every "AND" operations.

For generating coefficients of *s*-expanded polynomials, both methods give exactly the same results in terms of DDD sizes and path counts for the orders of coefficients computed. However, the existing method [8] fails for most large circuits – either it doesn't finish after 10 hours, or the DDD grows too large for the memory. On the other hand, the new method is able to get coefficients of the lowest 7 orders for circuits up to 140 nodes. We did not construct the coefficients for very large circuits due to excessive memory use of the coefficient DDDs beyond the memory in our Linux workstation.

8. CONCLUSION AND FUTURE WORKS

In this paper, we proposed a novel approach to constructing symbolic expressions in terms of DDD graphs for very large analog circuits. We formulate the DDD-based symbolic analysis process as a logic circuit synthesis problem and DDD can be constructed by a number of logic operations instead of Laplace expansion as before. The significance of our method is that the DDD construction time is no longer directly tied to the number of product terms of a determinant, but approximately to the size of the final DDD sizes, which makes the symbolic analysis problem more tractable than before in practice. Logic operation based approach was also used to construct specific coefficients of *s*-expanded polynomials of a determinant directly and implicitly.

Our experimental results have validated the proposed method and showed that the new method has great speedup over Laplace expansion based methods in both complex DDD and *s*-expanded DDD construction on small analog circuits, and is able to analyze larger analog circuits exactly than existing flat and hierarchical symbolic analyzers.

To enable best BDD construction process, a better ordering algorithm and dynamic variable ordering should be employed in the future.

9. REFERENCES

[1] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. on Computers*, pp. 677–691, 1986.

[2] G. Gielen and W. Sansen, *Symbolic Analysis for Automated Design of Analog Integrated Circuits*. Kluwer Academic Publishers, 1991.

[3] G. Gielen, P. Wambacq, and W. Sansen, "Symbolic analysis methods and applications for analog circuits: A tutorial overview," *Proc. of IEEE*, vol. 82, no. 2, pp. 287–304, Feb. 1994.

[4] G. H. Golub and C. F. V. Loan, *Matrix Computations*, 3rd ed. Baltimore, MD: The Johns Hopkins University Press, 1989.

[5] M. M. Hassoun and P. M. Lin, "A hierarchical network approach to symbolic analysis of large scale networks," *IEEE Trans. on Circuits and Systems I: Fundamental Theory and Applications*, vol. 42, no. 4, pp. 201–211, April 1995.

[6] P. M. Lin, *Symbolic Network Analysis*. Elsevier Science Publishers B.V., 1991.

[7] S. Minato, "Zero-suppressed bdds for set manipulation in combinatorial problems," in *Proc. Design Automation Conf. (DAC)*, 1993, pp. 272–277.

[8] C.-J. Shi and X.-D. Tan, "Canonical symbolic analysis of large analog circuits with determinant decision diagrams," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, no. 1, pp. 1–18, Jan. 2000.

[9] ——, "Compact representation and efficient generation of s-expanded symbolic network functions for computer-aided analog circuit design," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 7, pp. 813–827, April 2001.

[10] J. A. Starzky and A. Konczykowska, "Flowgraph analysis of large electronic networks," *IEEE Trans. on Circuits and Systems*, vol. 33, no. 3, pp. 302–315, March 1986.

[11] S. X.-D. Tan, W. Guo, and Z. Qi, "Hierarchical approach to exact symbolic analysis of large analog circuits," in *Proc. Design Automation Conf. (DAC)*, June 2004, pp. 860–863.

[12] X.-D. Tan and C.-J. Shi, "Hierarchical symbolic analysis of large analog circuits via determinant decision diagrams," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, no. 4, pp. 401–412, April 2000.

[13] J. Zhu and S. Calman, "Symbolic pointer analysis revisited," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2004.