Simulation of Switching Circuits using Transfer Functions

David Kebo Houngninou
Department of Computer Science and Engineering, SMU
Dallas, Texas, USA
Email: dhoungninou@mail.smu.edu

Mitchell A Thornton
Darwin Deason Institute for Cyber Security, SMU
Dallas, Texas, USA
Email: mitch@lyle.smu.edu

Abstract*—Simulation of complex hardware circuits is the basis for many EDA tasks and is commonly used at various phases of the design flow. State-of-the-art simulation tools are based upon discrete event simulation algorithms and are highly optimized and mature. Symbolic simulation may also be implemented using a discrete event approach, or other approaches based on extracted functional models. The common foundation behind modern simulation tools is that of a switching or Boolean algebraic model that may be augmented with timing information. Recently, an alternative foundational model for conventional digital electronic circuits has been proposed, based upon linear algebra rather than Boolean switching algebra where the circuits are modeled as transfer functions in the form of linear transformation matrices. We demonstrate that this model can be effectively used as the basis for a simulation methodology. Our approach is motivated by the need to develop a truly unified EDA tool for mixed signal circuit design. Currently, industrial tools such as SPECTRE use two different internal engines; a SPICE-like engine and a Verilog-like engine. Our method will allow us to represent mixed signal circuit elements as transfer functions. Spatial complexity is significantly reduced through the use of binary decision diagrams (BDD) to represent the transfer functions. A prototype implementation is used to generate experimental results and to illustrate the viability of the linear algebraic model as a basis for EDA applications.

Keywords: Switching Circuit Simulation, Symbolic Simulation, Transfer Function, Binary Decision Diagram

I. Introduction

The concept of a transfer function model for digital circuits is devised wherein the input stimulus and the output response are represented by an element in a finite-dimensioned Hilbert vector space. This work describes the use of our past results to implement and evaluate a prototype simulation tool. The prototype parses a structural netlist in Verilog and constructs the transfer matrix for the netlist in the form of a BDD. Constructing the transfer function of a structural circuit description can be accomplished by partitioning the netlist into a serial cascade of parallel stages, constructing the transfer matrices of each stage through a tensor matrix multiplication, and combining the stages using direct matrix multiplication [5]. The advantage of our simulation approach is that it supports symbolic simulation wherein any of the inputs, or subsets of the inputs can be assigned both binary values simultaneously. In one extreme, all possible input values can be symbolically simulated with one vector-matrix computation. In the other extreme, a single input assignment can be simulated with one vector-matrix product.

This paper is organized as follows. A background section briefly reviews the required theoretical concepts used to build the prototype simulator. The next section describes how the simulator is implemented including the relevant matrix-based models with the BDD-based algorithms employed to perform the computations. Following the implementation, we compare the performance of two simulation methods in terms of computation time and storage requirements.

II. Related Work

The transfer function concept as described in [6] provided the theoretical background for simulation and justification. The corresponding transformation from the input stimulus vector space to the output response vector space is given by a matrix. In [7] these theoretical results are further extended to cover non-binary switching circuits and to characterize the transfer functions representing switching circuits in spectral domains. To reduce the spatial complexity and improve the performance of applications based on the linear algebraic approach, we use binary decision diagrams (BDDs) to represent vectors and matrices. The implementation of the theory using BDDs is described in [5] where we provided algorithms for parsing a structural netlist into a BDD transfer function, and included required operations such as the inner product of vectors, the direct vector-matrix product, and the outer product of matrices. [5] also provided some experimental results for the generation of transfer functions as Binary Decision Diagrams and made a comparison of the compactness of the diagrams using variable ordering techniques such as sifting.

III. Building the prototype simulator

The first step involved in building our prototype consists of parsing the netlist to be simulated. The parser extracts information from a Verilog structural model in the form of a multi-level combinational logic circuit. Next, it tokenizes the netlist and extracts information such as the list of inputs, outputs, internal wires, and logic gates. Each gate and wire are assigned a unique ID number. After parsing, we identify fanouts by searching for gates with identical input nets. The parser splits the netlist into parallel stages or partitions representing subsets of the circuit. We can split the netlist into stages using levelization which consists of assigning level numbers to each gate according to its order of simulation. Each partition can count hundreds of logic gates and other topological features. Each partition is modeled in the form of a transfer function. In further steps, these functions are combined into a single overall monolithic transfer function.

A. Simulation using a matrix transfer function

A transfer function matrix T is obtained using a collection of transfer matrices for each structural element (Figure 2) that are reused as building blocks to construct the overall circuit transfer matrix. In this case, the gates are `AND`, `OR`, `XOR`, `BUF`, `NAND`, `NOR`, `XNOR`, `INV`. In addition to these gates, we also account for fanout and crossover features in the netlist. We treat fanouts as network elements since they also perform a transformation of the inputs. Crossovers are the topological intersection of conducting wires. The permutation of the order of wires affects the BDD variable ordering and interchanges the order of row vectors during simulation. We use a variable reordering transformation to align all variables in the stages containing crossovers.

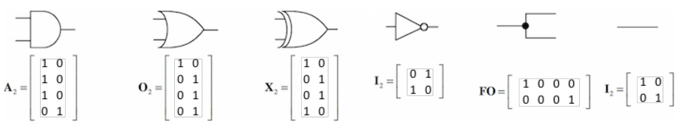

$A_2 = \begin{bmatrix}1&0\\1&0\\1&0\\0&1\end{bmatrix}$
$O_2 = \begin{bmatrix}1&0\\0&1\\0&1\\0&1\end{bmatrix}$
$X_2 = \begin{bmatrix}1&0\\0&1\\0&1\\1&0\end{bmatrix}$
$I_2 = \begin{bmatrix}0&1\\1&0\end{bmatrix}$
$FO = \begin{bmatrix}1&0&0&0\\0&0&0&1\end{bmatrix}$
$I_2 = \begin{bmatrix}1&0\\0&1\end{bmatrix}$

Fig. 1. Summary of six primitive operator matrices

We use the tensor product also called the outer or Kronecker product to compute the transfer function for a partition. This representation is suitable for systems composed of nets in a parallel arrangement; each net represents a matrix [4]. We can perform the Kronecker product $u \otimes v$ on each parallel element which is equivalent to the matrix multiplication $u \cdot v^{\mathsf{T}}$. Each partition transfer matrix requires $p - 1$ Kronecker products where p is the number of parallel elements. For example, the logic circuit in Figure 2 is composed of one `AND` gate and one inverter. This circuit is partitioned into a series of cascades. In this example, we identify three partitions θ_1, θ_2, θ_3. Because each partition is composed of a set of parallel elements, the signals on each parallel line must be combined into a single element in $\mathbb{H}^w$ where $\log_2 w$ is the number of parallel network signals. We build each of the partition matrices using the Kronecker product of each network element and multiply the partition matrices using a direct product to produce the monolithic transfer function matrix.

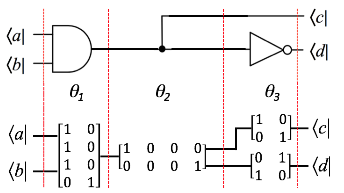

$\langle a|$
$\langle b|$
$\langle c|$
$\langle d|$
θ_1
θ_2
θ_3
$\langle a|$
$\langle b|$
$\begin{bmatrix}1&0\\1&0\\1&0\\0&1\end{bmatrix}$
$\begin{bmatrix}1&0&0&0\\0&0&0&1\end{bmatrix}$
$\begin{bmatrix}1&0\\0&1\end{bmatrix}$
$\begin{bmatrix}0&1\\1&0\end{bmatrix}$
$\langle c|$
$\langle d|$

Fig. 2. Sample circuit with corresponding partition matrices

$\langle a|$ and $\langle b|$ represent the input vectors. The intermediate matrices in partitions θ_1, θ_2, θ_3 perform a linear transformation of $\langle a|$ and $\langle b|$ to produce output vectors $\langle c|$ and $\langle d|$. Each partition is a transfer function on its own. In Figure 2, only cascade θ_3 has two network elements; therefore, we apply the Kronecker product to the nets in that cascade. A transfer function will require $m - 1$ direct product operations where m is the number of partitions. We perform the direct products by pair starting from the leftmost partition.

$$\mathbf{T}_{\theta_3} = \mathtt{I} \otimes \mathtt{Inv} = \left[\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right] \otimes \left[\begin{smallmatrix}0&1\\1&0\end{smallmatrix}\right] = \left[\begin{smallmatrix}0&1&0&0\\1&0&0&0\\0&0&0&1\\0&0&1&0\end{smallmatrix}\right]$$

$$\mathbf{T}_{\theta_1}\mathbf{T}_{\theta_2}\mathbf{T}_{\theta_3} = \left[\begin{smallmatrix}1&0\\1&0\\1&0\\0&1\end{smallmatrix}\right] \left[\begin{smallmatrix}1&0&0&0\\0&0&0&1\end{smallmatrix}\right] \left[\begin{smallmatrix}0&1&0&0\\1&0&0&0\\0&0&0&1\\0&0&1&0\end{smallmatrix}\right] = \left[\begin{smallmatrix}0&1&0&0\\0&1&0&0\\0&1&0&0\\0&0&1&0\end{smallmatrix}\right]$$

The output response of a logic network stimulated by an input $\langle x_q|$ and modeled by a transfer matrix $\mathbf{T}$ is denoted by $\langle f_q|$ and is computed using the equation: $\langle f_q| = \langle x_q|\,\mathbf{T}$. The inputs are denoted by an n-dimensional vector, $\langle x_i| \in \mathbb{H}^n$ and the outputs by a vector $\langle f_i| \in \mathbb{H}^m$. Figure 3 illustrates the linear transformation of two input values $\langle 1|$ and $\langle 1|$ by the same circuit.

$$\langle 1| \otimes \langle 1| = \langle 11_2| = \langle 3_{10}| = [\,0\;0\;0\;1\,]$$

$$\langle f_q| = \langle x_q|\,\mathbf{T} = [\,0\;0\;1\;0\,] = \langle 10_2| = \langle 2_{10}| = \langle 1| \otimes \langle 0|$$

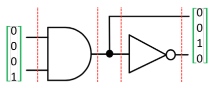
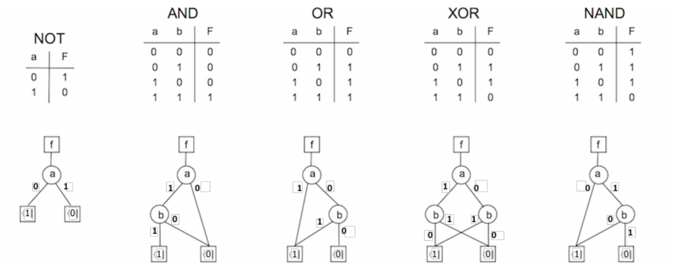


Fig. 3. Simulation using input vector $\langle 11|$

B. Simulation using Binary Decision Diagrams

A better approach for representing vectors and matrices is to use Binary decision diagrams [3]. Using the same approach presented above for matrices, we build a transfer function model using BDDs. In our implementation, we use a library of BDDs for each net using the CUDD package: a library for graphs manipulation. The table below shows the BDD representation of common logic gates.



NOT

a	F
0	1
1	0

AND

a	b	F
0	0	0
0	1	0
1	0	0
1	1	1

OR

a	b	F
0	0	0
0	1	1
1	0	1
1	1	1

XOR

a	b	F
0	0	0
0	1	1
1	0	1
1	1	0

NAND

a	b	F
0	0	1
0	1	1
1	0	1
1	1	0

Fig. 4. BDDs of logic gates

To model a multi-output circuit, we use algebraic decision diagrams. An algebraic decision diagram (ADD) is a decision diagram whose terminal nodes represent arbitrary integers other than just 0 and 1 [1]. The partition of a netlist is computed using the outer product (Kronecker) of all its network elements. To perform the Kronecker product of two ADDs, we use a polynomial. Let F be an ADD representing a function of n_1 variables and m_1 outputs. Let G be an ADD representing a function of n_2 variables and

m_2 outputs. The resulting Kronecker product $Z = F \otimes G$ is an ADD of $n_1 + n_2$ variables. For each path in ADD Z, we calculated the corresponding terminal node using the expression: $Z_{terminal} = 2^{m_2} \cdot F_{terminal} + G_{terminal}$. Using the $APPLY$ procedure developed by Bryant [2] with the above operator, we build the resultant graph Z. The procedure $APPLY$ requires two decision diagrams and an operator $\langle op \rangle$ as input operands and generates a reduced graph $F\langle op \rangle G$. The advantage of this procedure is that it provides a canonical and compressed tree as a result. The time complexity of the Kronecker product is $O(|F| \cdot |G|)$ where $|F|$ and $|G|$ represent the number of vertices in the graphs F and G respectively.

The direct product is also a necessary operation to obtain the overall transfer function. In the same way as for matrices, we multiply all partitions together starting from the leftmost one. The multiplication of two decision diagrams is a row transformation of the multiplier diagram by the multiplicand diagram. Since we formulated our transformation over the vector space, the values of the rows in the multiplicand ADD are used as pointers to the rows in the multiplier ADD. The following rules apply to the multiplication of two decision diagrams:

- The number of variables in the resulting decision diagram is equal to the number of variables in the multiplicand diagram
- The direct multiplication of two decision diagrams is non-commutative
- The direct multiplication of decision diagrams is associative

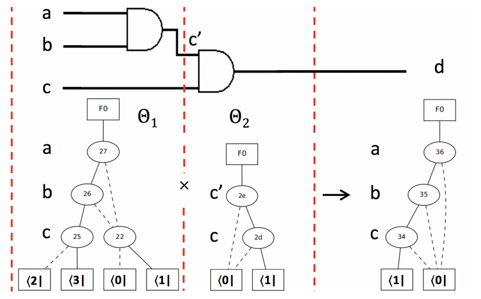


Fig. 5. Decision diagrams multiplication (matrix-by-matrix)

Figure 5 shows the partition cuts of a circuit composed of two AND logic gates. This circuit is parsed into two cascades. The first cascade is composed of an AND gate and a wire. It is represented by an ADD of three variables a, b, c and four terminal constants $2, 3, 0, 1$. The second cascade is composed of an AND gate. It is represented by an ADD of two variables c' and c and two terminal constants 0 and 1. Both partitions are multiplied together to produce an ADD of three variables and two terminal constants 0 and 1. The result represents the transfer function of the logic circuit and is isomorphic to the truth table of a 3-input AND. Once we obtain the transfer function, we can use it to perform simulation on any variable assignment. For three input values $a = 1$, $b = 1$ and $c = 0$, we get the following input vector: $\langle 1| \otimes \langle 1| \otimes \langle 0| = \langle 110_2| = \langle 6_{10}|$.

Figure 6 illustrates the multiplication of the input vector $\langle 6|$ by the transfer function.

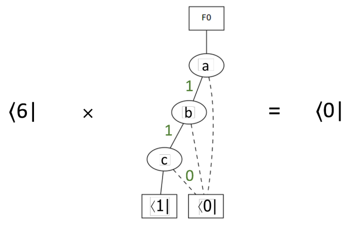


Fig. 6. Simulation using input vector $\langle 110|$ (vector-by-matrix)

The following algorithm includes two cases: the multiplication of an ADD by an ADD (matrix-by-matrix), and the multiplication of a constant by an ADD (vector-by-matrix). n_1 is the number of variables in the multiplicand F and n_2 is the number of variables in the multiplier G with the product denoted as Z.

Algorithm 1: Multiplication of two decision diagrams

```
  Input: Node $F$ and Node $G$
  Output: ADD $Z$ representing the resultant graph $F \times G$
1 if $n_1$ is equal to 0 then
      ▷ F is a constant node
2     $Z \leftarrow$ terminal node of $G$ for variable assignment $F$
3 else if $n_1$ is greater than 0 then
4     foreach paths in $F$ from root to terminal do
5         $Z_{paths}[i] \leftarrow$ variable assignment $F$
          $Z_{terminals}[i] \leftarrow$ terminal node of $G$ for variable assignment $F$
6     Build ADD $Z$
7 return $Z$;
```

In our prototype simulator, we experiment with two approaches for computing the output response. In the first approach, we formulate the overall circuit transfer function in the form of a single graph that we refer to as the "monolithic ADD". The second method omits the step of computing the transfer function as a block and instead retains an array of multiple ADDs where each represents the transfer function for an individual serial stage of the partitioned netlist.

C. Simulation using a monolithic transfer function

The monolithic method consists of formulating the overall circuit in the form of a single ADD. This method uses the ADD-by-ADD multiplication algorithm to combine all the partitions of the netlist into a single diagram, then multiplies it with the input stimulus vector to obtain the output response. The advantage of this method is that we can represent the entire function as one diagram and simulate for all input combinations in a single iteration. The downside of building a monolithic transfer function is potentially higher memory usage. The spatial complexity for multiplying two partitions is $O(|F| \cdot |G|)$.

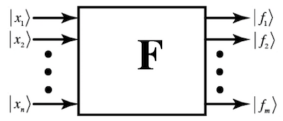


Fig. 7. A transfer function framework for F (Monolithic method)

D. Simulation using an array of transfer functions

Rather than building the entire transfer function, this method performs the simulation incrementally starting from the primary inputs. The array method consists of performing multiple vector-matrix multiplications over each partition to obtain the output response. The benefit of this technique is that we only build one ADD at a time for each cascade, and we can free up the nodes of previously used cascades after each iteration. Since we use only one partition at a time, building the output response incrementally is beneficial for reducing memory usage. Nodes from previous cascades are gradually dereferenced. In contrast to the monolithic method, a netlist represented by k partitions requires k vector-matrix multiplications to obtain each output response.

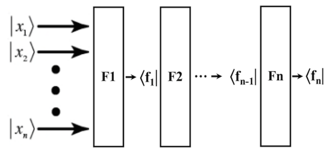


Fig. 8. A transfer function framework for F (Array method)

IV. Experimental results

To evaluate the two approaches for the prototype simulator, we use a set of benchmark circuits as input to the simulator with a randomly generated set of test vectors. The results consider both the peak memory usage and the simulation computation time.

TABLE I
SIMULATION OUTPUT RESPONSE (MONOLITHIC METHOD)

Benchmark	Inputs/ Outputs	# of partitions	# of nodes	Memory (MB)	Time to build partitions (ms)	Time to build BDD (ms)	Time to simulate (ms)
xor5.v	5/1	6	11	8.77	0.44	2.67	0.01
c17.v	5/2	12	12	8.90	0.57	4.88	0.02
majority.v	5/1	12	9	8.98	1.09	5.10	0.01
test1.v	3/3	16	10	8.92	0.86	5.49	0.01
rd53.v	5/3	18	21	9.29	1.21	10.48	0.04
con1.v	7/2	14	15	19.07	2.14	175.67	0.01
radd.v	8/5	28	109	19.12	4.91	296.04	0.03
rd73.v	7/3	24	71	19.34	5.56	76.96	0.01
mux.v	21/1	26	145	33.77	7.61	43.47	0.01
c432.v	36/7	57	451	41.08	240.60	945.89	0.08
c499.v	41/32	16	442	43.14	246.50	850.11	0.09
c880.v	60/26	67	895	67.90	1412.62	6580.10	0.21
c5315.v	178/123	80	1286	83.47	3150.11	7783.62	0.37
c2670.v	233/140	99	1560	97.01	6521.43	8195.09	0.58

TABLE II
SIMULATION OUTPUT RESPONSE (ARRAY METHOD)

Benchmark	Inputs Outputs	# of partitions	Memory (MB)	Time to build partitions (ms)	Time to build BDDs (ms)	Time to simulate (ms)
xor5.v	5/1	6	8.70	0.44	0.17	0.01
c17.v	5/2	12	8.79	0.57	0.70	0.03
majority.v	5/1	12	8.84	1.09	1.01	0.04
test1.v	3/3	16	8.81	0.86	0.84	0.03
rd53.v	5/3	18	9.06	1.21	2.70	0.07
con1.v	7/2	14	18.82	2.14	65.11	0.44
radd.v	8/5	28	21.08	4.91	195.43	0.54
rd73.v	7/3	24	11.72	5.56	21.37	0.19
mux.v	21/1	26	16.19	7.61	429.69	1.90
c432.v	36/7	57	18.42	240.60	619.09	2.78
c499.v	41/32	16	17.22	246.50	1150.11	2.01
c880.v	60/26	67	25.42	1412.62	7100.60	5.21
c5315.v	178/123	80	37.17	3150.11	8752.80	9.75
c2670.v	233/140	99	49.01	6521.43	9450.20	12.87

The tables above contain timing data, the total number of nodes in the transfer function and memory usage for both the monolithic method and the array method. The motivation for comparing these two approaches is that the monolithic ADD is larger and requires more memory for representation but only requires a single vector-matrix product computation to obtain the output response. In contrast, the array of ADDs method will result in less required storage but will require k vector-matrix computations to compute an output response vector. Using the array method, it is not necessary to compute the entire array of ADDs since only a single we compute one partition at a time. For both methods, crossovers between partitions are handled by reordering variables accordingly in the corresponding ADD. As observed in the results, the monolithic approach generally required more memory but resulted in faster simulation times whereas the partition array method reduced memory usage at the expense of simulation time.

V. Conclusion

We have described how we can use the theory in [6] to implement a simulator. Two versions of the simulator were implemented with one optimizing runtime and the other optimizing memory usage. In order to make use of the theoretical results of [6] in a practical manner, ADDs are used to represent the matrices and vectors. A new tensor multiplication algorithm was formulated as an operation over matrices represented as ADDs and was shown to be very efficient in that it required only the traversal of a single path in each of the two operands ADDs. This tensor multiplication algorithm enabled the two candidate simulation methods to have reasonable and competitive runtimes and memory usage statistics. These results indicate that the linear algebraic theory can be used as a practical and reasonable alternative to conventional switching algebra models for digital circuit EDA tools.

References

[1] R Iris Bahar, Erica A Frohm, Charles M Gaona, Gary D Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. Algebric decision diagrams and their applications. *Formal methods in system design*, 10(2-3):171–206, 1997.

[2] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, C-35(8):677–691, Aug 1986.

[3] Edmund M Clarke, Masahiro Fujita, and Xudong Zhao. Multi-terminal binary decision diagrams and hybrid decision diagrams. In *Representations of discrete functions*, pages 93–108. Springer, 1996.

[4] Luca De Alfaro, Marta Kwiatkowska, Gethin Norman, David Parker, and Roberto Segala. *Symbolic model checking of probabilistic processes using MTBDDs and the Kronecker representation*. Springer, 2000.

[5] D. K. Houngninou and M. A. Thornton. Implementation of switching circuit models as transfer functions. In *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 2162–2165, May 2016.

[6] Mitchell Thornton. Simulation and implication using a transfer function model for switching logic. *IEEE Transactions on Computers*, PP, February 2015.

[7] Mitchell A. Thornton. *Modeling Digital Switching Circuits with Linear Algebra*. Morgan & Claypool Publishers, 2014.