Fast Functional Simulation Using Branching Programs

Pranav Ashar
CCRL, NEC USA
Princeton, NJ 08540

Sharad Malik
Princeton University
Princeton, NJ 08544

Abstract

This paper addresses the problem of speeding up functional (delay-independent) logic simulation for synchronous digital systems. The problem needs very little new motivation – cycle-based functional simulation is the largest consumer of computing cycles in system design. Most existing simulators for this task can be classified as being either *event driven* or *levelized compiled-code*, with the levelized compiled code simulators generally being considered faster for this task. An alternative technique, based on *evaluation using branching programs*, was suggested about a decade ago in the context of switch level functional simulation. However, this had very limited application since it could not handle the large circuits encountered in practice. This paper resurrects the basic idea present this technique and provides significant modifications that enable its application to contemporary industrial strength circuits. We present experimental results that demonstrate up to a 10X speedup over levelized compiled code simulation for a large suite of benchmark circuits as well as for industrial examples with over 40,000 gates.

1 Introduction

Design validation by simulation is a key step in the design cycle of digital systems. It is also one of the most time consuming. Each time a design is iterated, it must be re-simulated until a satisfactory confidence level is achieved. Simulation is performed at various levels of abstraction during the design process. We address the problem of speeding up delay-independent (i.e. purely functional) cycle-based logic simulation of synchronous digital circuits. At this level, the simulator is required to determine the output sequence produced by the circuit for a sequence of input vectors, independent of the delays associated with the gates and wires. In effect, the simulator determines the output vector for each input vector by evaluating the Boolean equations associated with each gate in the circuit. The circuit is assumed to possess feedback through flip-flops. Therefore, the circuit can be evaluated for the next input vector only after the evaluation for the current input vector is complete. This type of simulation where the input vectors must be simulated sequentially in this manner is said to be cycle-based sequential simulation.

Two techniques have traditionally been applied for logic simulation at this level:

- event-driven simulation, e.g. [16, 4]
- levelized compiled-code (LCC) simulation, e.g. [12, 9, 8, 17]

1.1 Evaluation Using Branching Programs

An alternative technique, presented by Cerny [7], relies on the use of decision diagrams to derive branching programs for function evaluation. This work was originally done for switch-level functional simulation but has an obvious application to logic level simulation. A decision diagram for a Boolean function evaluates the function by a sequence of decisions (typically two-way branches), with each decision based on the value of one of the variables that the function depends on. Decision diagrams were first introduced by Akers [2] and they shot into prominence with the introduction of the reduced ordered binary decision diagrams (ROBDDs or just BDDs) by Bryant [5]. Figure 1 shows an example of a decision diagram for the function $x1x2 + x3$.

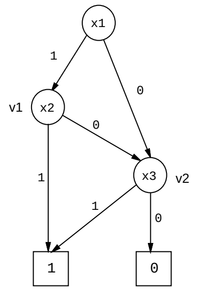


Figure 1: Example Decision Diagram for $f = x1x2 + x3$

```
    if(x1) then goto v1;
   else goto v2;
v1: if(x2) then return 1;
   else goto v2;
v2: if(x3) then return 1;
   else return 0;
```

Figure 2: Example Branching Program for $f = x1x2 + x3$

Several researchers have pointed out the straightforward isomorphism between a decision diagram and a branching program (e.g. [11]). A branching program evaluates its function using a sequence of multi-way branches, with each branch depending on the value of one of the arguments of the function. The branching program for the decision diagram example in Figure 1 is shown in Figure 2. This is just a direct transformation from the decision diagram and has not been optimized in any way.

Since a specialized program is generated for each circuit, this method is a form of compiled-code simulation. The main advantage of this method is that the complexity of simulating a vector is proportional to the number of input arguments of the function, since each one needs to be examined only once to determine the function value. This seems like a big win, since the number of inputs is typically very much less than the number of gates in the circuit, which is the complexity of levelized compiled-code simulation. However, in spite of this intuitive elegance, this method did not see practical success due to the following factors:

- It was unclear how the approach could be extended from a single output function to multiple output functions.
- The size of the decision diagrams grows very quickly, resulting in programs that can either not be stored, or not be easily compiled, or result in memory thrashing.

1.2 Proposed Approach

Our proposed approach resurrects the main ideas in using decision diagrams/branching programs to evaluate Boolean functions and provides significant modifications that enable it to overcome the previous limitations. These can be summarized as follows:

- Instead of using a decision diagram for each single-output function in the circuit, we use decision diagrams for characteristic functions of multi-output functions [6].

- Instead of a single decision diagram for the entire circuit, we use a set of decision diagrams. Between them, the characteristic functions in this set compute all the outputs, i.e. the outputs are partitioned among the characteristic functions. While Cerny did mention the use of a set of decision diagrams, no algorithm for this was provided, nor any application shown for large circuits. *The key feature of our partitioning technique is that it is done dynamically based on the size of the decision diagrams.* Thus, it is guaranteed never to result in memory blow up. As soon the memory requirements reach a threshold, a new characteristic function is created.

As a consequence of this partitioning, this method encompasses both levelized compiled-code, as well as branching program based evaluation. It reduces to the former when the each partition corresponds to a single gate, and to the latter when the there is a single partition that contains the entire circuit. Intuitively we feel that this method finds the right partitioning along this spectrum for a given circuit. Practically, the results demonstrate a 10X speed up over levelized compiled-code simulation for many circuits from a large suite of benchmarks.

2 Simulation Using BDD-Based Characteristic Functions

The reader is referred to [5] for an introduction to the Binary Decision Diagram (BDD) data structure.

2.1 Single Output Circuits

Consider first the problem of simulating a single-output circuit. The support of a function is defined as the set of input variables that the function depends on. Let the number of input variables in its support be #I, and let the number of gates be #G. Further, let us assume that each gate implements a basic Boolean operation such as NOT, AND or OR. If the BDD for the circuit is available, the output can be determined by tracing the unique path corresponding to the input vector from the root of the BDD to a leaf. The value of the output is the 0 or 1 value associated with the leaf reached. At most as many vertices of the BDD are encountered along the path as #I.

Compared to conventional simulation techniques in which the Boolean equations associated with gates are evaluated at run time (total evaluation time is therefore proportional to #G), the decision diagram approach is at least asymptotically much faster since #I $<<$ #G in typical circuit circuits. We recognize the fact that asymptotic complexity may not always be the best measure of the practical utility of an algorithm, the constants involved in a particular implementation can strongly influence the practical utility. However, the asymptotic complexity does serve as a useful first-level complexity measure.

The downside is that the BDD representation of a circuit is usually larger than a representation consisting of arbitrary gates. Also, it does not permit the same kind of pipeline and memory hierarchy utilization as LCC simulation. The ideal situation would of course be to store the truth table for the circuit and read off the output in constant time for the applied input vector. Storing the truth table is obviously infeasible for a circuit with more than a handful of inputs. A BDD representation is usually much more compact than a truth table. In a sense, the use of decision diagrams for evaluating the output represents a middle ground between evaluating the gates at run time and storing the truth table for the circuit - they are likely to be faster to evaluate than LCC simulation but at the cost of greater memory requirement, and they are much more compact than a truth table but require some computation at run time.

2.2 Multiple-Output Circuits

It is unlikely that any circuit encountered in practice will have a single output. We now consider the general case, where more than one output is present. Let the number of outputs in the circuit be #O. In the naive approach we can evaluate each output separately using its decision diagram as described in Section 2.1. The total running time in the worst case (when all the outputs depend on all the inputs) is proportional to #I $\times$ #O, since #I evaluations may be needed for each output. This clearly does not compare favorably with conventional simulation techniques since #I $\times$ #O is typically much larger than the number of gates (#G).

We would like to avoid the product of #I and #O in the asymptotic complexity if possible. An almost obvious solution here is to try and simultaneously evaluate the multiple outputs in the decision diagram. That way, only #I evaluations will be needed to accomplish this. (A further #O operations may be needed to read the individual outputs from the result of the multiple-output evaluation.) This is accomplished by using BDD-based characteristic function representation of the circuit to determine the values of the multiple outputs for the applied input vector. Characteristic functions were first introduced by Cerny [6] and have since then seen much use in the synthesis and verification of digital systems.

Definition 2.1 *Let E be a set and $A \subseteq E$. The characteristic function of A is the function χ_A: $E \rightarrow \{0, 1\}$ defined by $\chi_A(x) = 1$ if $x \in A$, $\chi_A(x) = 0$ otherwise[15].*

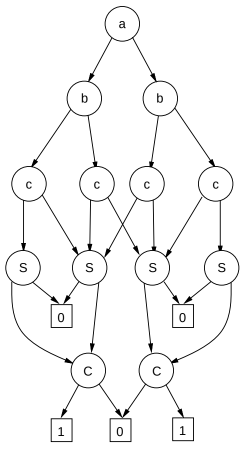
Thus, the characteristic function is a Boolean function representation of a subset of some universal set. The particular universal set of interest to us is the set of all possible $2^{\#I+\#O}$ input-output combinations for a given circuit. The subset of interest is the set of $2^{\#I}$ *valid* combinations. A combination is said to be valid if the output vector in the combination is the output produced by the circuit when the input vector in the combination is applied to it. In essence, the characteristic function captures the complete functionality of the multi-output circuit.

How do we use the characteristic function to evaluate the multiple outputs in run time much smaller than #I $\times$ #O? The characteristic function of a multiple-output circuit can be represented by a BDD with #I + #O variables. Figure 3 shows an example of the BDD for the characteristic function of a 1 bit full adder. a and b are the data inputs, c is the carry in, S is the sum ($a \oplus b \oplus c$) and C is the carry out ($ab + ac + bc$). Each path to a 1 leaf in this BDD corresponds to a single valid input-output combination.

That is not enough for multi-output evaluation since the output vector is not known *a priori* and we need to *determine* it. One solution, albeit not a very good one, is to impose a variable ordering on the BDD which requires that all the output variables appear only after all the input variables have appeared. Since each input vector produces a unique output vector, once all the input values have been fixed, there will be a unique path from that point to a 1 leaf through the vertices corresponding to the output variables. Therefore, to determine the output vector one must first trace the sub-path in the BDD corresponding to the input values; from that point, one just needs to trace the unique path to the 1-leaf. Along the way, one reads off the values taken by the output variables. These values are the outputs produced by the circuit. For example, in Figure 3, for the input vector $< a = 1, b = 0, c = 1 >$, corresponding circuit outputs are $< S = 0, C = 1 >$. The evaluation time is proportional to the number of vertices encountered on the traced path. Since there are #I + #O variables in the characteristic function, the evaluation time is bounded by #I + #O. In general, #I + #O can be expected to be much smaller than #G in the typical circuit. Therefore, the use of the BDD based characteristic function representation allows us to simulate circuits much faster than conventional simulation. If we just look at the asymptotic complexity, this solution looks pretty good. However, this is still not good enough.

2.2.1 Variable Ordering

It is well known that the size of a BDD is usually very sensitive to the variable ordering chosen. In the solution just suggested, we imposed an ordering in which output variables are allowed to appear



The right edges are the 0 edges and the left edges are the 1 edges.

Figure 3: BDD for the Characteristic Function of a Full Adder

only after the input variables. This is a very bad ordering and as a consequence this solution is impractical. To see why this is so, let us assume we use this ordering. Now consider all BDD nodes such that they correspond to an output variable, and their parents correspond to input variables. Let this set be N. The BDDs rooted at each $n \in N$ contain only the output variables. The cardinality of N is the number of distinct output combinations that the circuit can produce. This is typically exponential in the number of circuit outputs. This exponential growth severely limits the number of outputs that this method can handle.

We can do better. Let us relax the ordering condition that the output variables appear only after all the input variables in the BDD. To maintain the #I + #O run time complexity, we only need to ensure that we never have to follow both the THEN (or 1) and ELSE (or 0) edges of a BDD node corresponding to an output variable. Whenever an output variable is encountered in a path of the BDD, its value must already have been determined by the values assigned to the input variables seen until that point. To ensure this, we just have to guarantee that an output variable appears on any path only after all the input variables in *its* support set have appeared. This relaxation of ordering constraints allows us to interleave the output variables with the input variables. While in the worst case this still does not provide any guarantees on the size of the BDD, at least we have avoided the more obvious exponent in the size with all the output variables ordered after all the input variables. Fortunately, it turns out that interleaving the output variables among the input variables is an excellent variable ordering heuristic for BDDs of characteristic functions [15].

The ordering algorithm (from [15]) operates in two steps: First a good ordering of the output variables $\{y_i\}$ is obtained as described below. Subsequently, the input variables in the support of each of the output functions are ordered individually according to the algorithm in [10]. Let the output functions associated with the output variables $\{y_i\}$ be denoted by $\{f_i\}$. Let the support of each function be denoted by $supp(f_i)$. In the final interleaved ordering, the output variables follow immediately after their support as follows: $supp(f_1), y_1, \ldots, supp(f_n) - \cup_{1 \leq i \leq n-1} supp(f_i), y_n$. The ordering of the output variables is chosen so that the following cost function is minimized:

$$cost(\sigma) = \sum_{1 \leq i \leq n} |\cup_{1 \leq j \leq i} supp(f_{\sigma_j})|$$

Given a set A, $|A|$ denotes the cardinality of the set. Finding a permutation that minimizes this cost function exactly is not feasible. In practice, the greedy heuristic of looking ahead a few (2 or 3) levels, trying out all possible permutations up to those few levels and choosing that one with minimum cost seems to work well. In effect, the outputs are ordered so that the outputs with many support variables in common are bunched together. It is this ordering heuristic

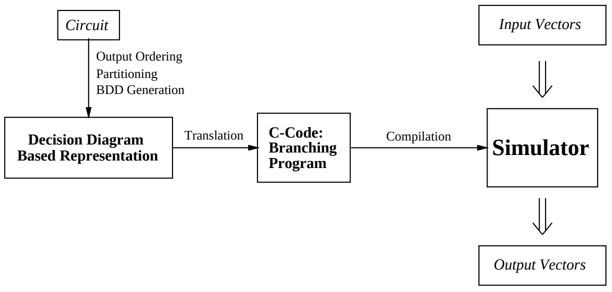


Figure 4: Overall Flow of Decision Diagram Based Simulation

that we will be using to construct the BDDs for our characteristic functions.

It has been demonstrated recently that dynamic variable ordering, *i.e.* reordering the variables while building the BDDs, works well in practice [13]. In our case also, once we have obtained the output and input variable ordering heuristically, we can try to improve upon it during BDD creation using the dynamic variable ordering techniques. Note that we can only allow the input variables to be moved towards the root and the output variables to be moved towards the leaves.

Finally, the variable ordering on different paths in the decision diagram need not be identical as long as we ensure that each input-output combination corresponds to a single path in the decision diagram and that the output variables appear only after their support variables have appeared. Decision diagrams that allow different orderings along different paths are called free-BDDs. They are potentially smaller in size than BDDs, but are not as efficient to manipulate. Their use for generic circuits has been investigated in the recent past [1, 11]. If free-BDD manipulation software is available, it can potentially be used to reduce the size of the characteristic function representation. Dynamic variable ordering and the use of free-BDDs have not been implemented yet.

2.3 Multiple Characteristic Functions

Ideally, for maximum speed up, we would like to represent the entire circuit with a single BDD-based characteristic function. In practice, that may not be possible for a variety of reasons - a function requiring a BDD provably exponential in size for any variable ordering may exist in the circuit, different outputs of the circuit may require different variable orderings, the circuit may have an unmanageably large number of inputs and outputs, or a good variable ordering for the circuit may not be easy to find.

Fortunately, the application of BDDs to simulation is one that degrades gracefully when such a situation is encountered. The solution is to represent the circuit by a set of characteristic functions, X, such that all the outputs can be evaluated using this set. In this process, we possibly create additional outputs and inputs which correspond to intermediate signals in the circuit.

X is then levelized, where the level of a characteristic function, χ_i, is defined as the number of characteristic functions that must be evaluated in order to evaluate χ_i. The entire circuit is then simulated by evaluating the χ_is in the order of increasing levels. The levelizing ensures that the values for the inputs to a given χ_i will be available when it is to be evaluated. This strategy comes at the expense of a loss in the speed up. In general, the greater the cardinality of X, the greater is the loss in speed up. But, as we observe in practice, even a simple heuristic for generating these sets is sufficient to keep number of characteristic functions that need to be evaluated small. Figure 4 shows the overall flow of the compiled code simulation package.

An exact solution to the partitioning problem would try to generate the minimum number of χ_is while ensuring that these can be built with available memory. This is obviously a difficult problem and we do not attempt to solve it exactly. We find that the following strategy of adding additional inputs and outputs on demand works well in practice:

1. Order the outputs according to the clustering algorithm described in Section 2.2.1.
   (a) Order the input variables for this output order.
2. Build BDD for the next output in the order.
   (a) If BDD construction is not possible due to memory limits, select a set of nodes (called a cut) in the transitive fanin of the output so that BDD size at each node in the cut is less than some threshold value. This cut satisfies the property that the nodes in the cut are not in the transitive fanin of each other, i.e., no node function for any of the cut nodes can directly depend on the node function of another cut node.
   (b) Order only the new outputs according to the clustering algorithm described in Section 2.2.1.
   (c) Build BDD for the next new output in the new-output order. This step is guaranteed to finish by virtue of Step 2(a).
   (d) Add this new output variable to the current characteristic function.
      i. If this step is not possible because the resulting BDD size exceeds the allowed threshold, start a new characteristic function for the remaining new outputs.
   (e) Return to Step 2(c) if some new output has not been incorporated into a characteristic function. Else continue to 2(f).
   (f) Add the new inputs corresponding to the cut made in Step 2(a) to the set of inputs.
   (g) Restart at Step 1 with the updated set of inputs, and a new characteristic function.
3. Add the output variable to the current characteristic function.
   (a) If Step 3 is not possible because the resulting BDD size exceeds the allowed threshold, start a new characteristic function and add the output to it.
4. Repeat until all outputs have been included in some characteristic function.

If a new output is in the transitive fanin of some original output, then these two outputs can never occur in the same characteristic function because the value of the output in the transitive fanin is needed to evaluate the value of the other output. Steps 2(b) - 2(g) ensure this by keeping the newly generated outputs and the original outputs that have yet to be processed in different characteristic functions.

2.4 Non-Binary Valued Simulation

The previous sections have focussed on the use of binary decision diagrams to generate branching programs for use in simulation. The key assumption made here is that the final simulation will be binary-valued, i.e. all signals will have values either 0 or 1. This is not always true, in practice multiple logic values are used in simulation. The non-binary values that are used are:

- X: An unknown value, representing either 0 or 1.
- Z: A high impedance value, representing neither 0 nor 1.

One solution is is to encode the four logic values of each input by using two binary-valued variables. In practice, this results in larger BDDs and a significant loss of speedup.

A more practical solution is to use an event-driven-simulator front end to drive out the X's arising from unknown initial states on the memory elements. Once the X's have been driven out, 2-valued functional simulation can be used. Z's need to be simulated in order to detect contention between bus-drivers. In the cycle-based delay-independent simulation framework, this contention can be detected by analyzing the Boolean values on the controlling inputs of the tristate-buffers. In fact, some commercial LCC simulators also follow this policy.

Circuit	# I	# O	# G	# L (unopt)	# L (opt)
s420	35	18	196	336	199
s510	25	13	211	424	313
s838	67	34	390	670	447
s1494	14	25	647	1393	836
s9234	247	250	5597	7971	2507
s13207	700	790	7951	11165	3394
s15850	611	684	9772	13645	4379
s35932	1763	2048	16353	28269	11257
s38417	1664	1742	22179	32028	17404
s38584	1464	1730	19253	32756	15880
NA37	476	457	1766	3501	3658
MAIN	1133	1106	21698	35392	8525
tur13	2131	2304	38182	63556	39757

I/O/G/L: Primary Inputs/Primary Outputs/Gates/Literals in the un-optimized (unopt) and the SIS optimized (opt) networks.

Table 1: Statistics of the Circuits Used in Our Experiments

3 Experimental Results

We have tried our simulation approach on all circuits from the combinational and sequential ISCAS benchmark suites, and three large industrial circuits. We present two-valued simulation results on some of the larger circuits from the set. The largest circuit in the set, `tur13`, has about 40,000 gates and 65,000 literals in its un-optimized form. We demonstrate through these results that considerable reductions in simulation time can be obtained using our approach. The BDD package distributed with SIS [14] was used in these experiments to construct the branching programs. For purpose of comparison we implemented a levelized compiled code simulator (for two-valued simulation). The compiler `cc` distributed with `SunOS Release 4.1.3` was used to compile the branching programs as well as the levelized gate-netlists. The optimization option `-O1` was used during all compilation. All run times are on a `SPARC 2` with 128 Meg of main memory. Relevant data about the circuits is provided in Table 1. **# L (unopt)** corresponds to the number of literals in the original unmodified gate netlists. **# L (opt)** corresponds to the number of literals after optimization for literal count using SIS. The purpose of the optimization is to extract common sub-expressions which help in speeding up the simulation.

Simulation times for evaluating one million random vectors are reported in Table 2. These are raw simulation times and do not include the time taken to read and write the vectors. The column **BBCF** (stands for BDD-based characteristic function) contains the simulation times using our approach. The column **# Partitions** indicates the number of partitions that were required in the BDD-based simulation. A 1 indicates that a single BDD-based characteristic function was required. LCC simulation times can be considerably reduced by optimizing the circuit for literal count prior to simulation. LCC simulation times for the optimized netlists are reported in column **LCC(opt)**. To obtain these times, the optimized netlists were first levelized, with level of a gate being defined as the largest number of gates on any path from a primary input to it. The levelized netlists were translated into C-code with additional code added for I/O. This code was then compiled into an executable binary. The corresponding speedups, defined as (LCC simulation time)/(BBCF simulation time), are provided in the column **Speedup**. The total times required for generating the simulation binaries are indicated in the column **Prep. Time**. The sizes of the simulation binaries are indicated in the column **Binary Size**.

4 Conclusions

We have presented a new technique for functional simulation that uses branching programs. We have demonstrated multiple factors of speed up using this technique over LCC simulation. The main algorithmic contributions of this paper are as follows:

Circuit	Simulation Time (sec)		Speedup	# Partitions	Binary Size (KB)		Prep. Time (sec)	
	LCC(opt)	BBCF	LCC/BBCF		LCC(opt)	BBCF	LCC(opt)	BBCF
s420	31.3	10.4	3.0	1	2	6	32	32
s510	44.6	8.2	5.4	1	3	4	32	32
s838	85.4	18.6	4.6	1	6	31	40	72
s1494	136.9	11.5	11.9	1	6	8	40	40
s9234	617.0	378.8	1.6	10	69	700	96	944
s13207	2011.0	944.0	2.1	9	298	1200	192	892
s15850	2406.0	983.0	2.4	18	216	2010	184	1836
s35932	7120.0	2018.0	3.5	4	1056	1109	520	800
s38417	10672.0	3598.0	3.0	64	1354	8750	632	4136
s38584	9254.0	2770.0	3.3	40	959	9298	560	6400
NA37	1489.0	403.7	3.7	8	250	1100	144	768
MAIN	5123.0	1081.0	4.7	20	436	5116	352	2960
tur13	29653.0	3425	8.7	64	3498	9778	1224	9864

Simulation time is the time for evaluating one million vectors. It is the raw simulation time and does not include time for reading and writing vectors. All abbreviations are explained in text.

Table 2: Comparing the Raw Simulation Speeds of Conventional LCC to BBCF

- Demonstrating how BDDs for characteristic functions can be used to handle multiple-output functions for simulation. While it is known that characteristic functions are a useful representation for multiple-output functions, it is not obvious how a BDD for a characteristic function can be used for simulation. Not all variable orders permit simulation, an output variable must appear in the order only after its support. In addition, the naive ordering of all output variables at the end needs to be avoided since it results in memory explosion.

- Demonstrating how a set of characteristic functions can be dynamically generated based on memory usage.

There is probably room for considerable improvement. A compiler customized for generating machine code from branching programs could potentially speed up compile times considerably, making this method much more attractive. There is also room for improving the partitioning algorithm. The use of dynamic variable ordering can potentially speed up the simulation further by reducing the sizes of the simulation binaries and reducing the number of partitions.

References

[1] A. Ghosh A. Shen, S. Devadas. Probabilistic construction and manipulation of free boolean diagrams. In *Proceedings of the International Conference on Computer-Aided Design*, pages 544–549, November 1993.

[2] S. B. Akers. Binary decision diagrams. In *IEEE Transactions on Computers*, volume C-27, pages 509–516, June 1978.

[3] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. MIS: A multiple-level logic optimization system. In *IEEE Transactions on CAD*, volume C-6, pages 1062–1081, November 1987.

[4] M. A. Breuer and A. D. Friedman. *Diagnosis and Reliable Design of Digital Systems*. Computer Science Press, Inc., 1976.

[5] R. Bryant. Graph-based algorithms for Boolean function manipulation. In *IEEE Transactions on Computers*, volume C-35, pages 677–691, August 1986.

[6] A. Cerny. An approach to unified methodology of combinational switching circuits. In *IEEE Transactions on Computers*, volume 27(8), August 1977.

[7] A. Cerny and J. Gecsei. Simulation of MOS circuits by decision diagrams. In *IEEE Transactions on Computer-Aided Design*, volume C-4, pages 685–693, October 1985.

[8] M. Chiang and R. Palkovic. LCC simulators speed development of synchronous hardware. In *Computer Design*, pages 87–91, 1986.

[9] N. Ishiura, H. Yasuura, T. Kawata, and S. Yajima. High-speed logic simulation on a vector processor. In *Proceedings of the International Conference on Computer-Aided Design*, pages 119–121, November 1985.

[10] S. Malik, A. R. Wang, R. Brayton, and A. Sangiovanni-Vincentelli. Logic Verification using Binary Decision Diagrams in a Logic Synthesis Environment. In *Proceedings of the International Conference on Computer-Aided Design*, pages 6–9, November 1988.

[11] Christoph Meinel. *Modified branching programs and their computational power*. Springer-Verlag, 1989.

[12] G. F. Pfister. The Yorktown Simulation Engine: Introduction. In *The Proceedings of the Design Automation Conference*, pages 51–54, June 1982.

[13] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proceedings of the International Conference on Computer-Aided Design*, pages 42–47, November 1993.

[14] E. M. Sentovich, K. J. Singh, C. Moon, H. Savoj, R. K. Brayton, and A. Sangiovanni-Vincentelli. Sequential circuit design using synthesis and optimization. In *Proceedings of the International Conference on Computer Design*, 1992.

[15] H. Touati, H. Savoj, B. Lin R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using BDDs. In *Proceedings of the International Conference on Computer-Aided Design*, pages 130–133, November 1990.

[16] E. Ulrich. Exclusive simulation of activity in digital networks. In *Communications of the ACM*, volume 13, pages 102–110, February 1969.

[17] L. T. Wang, N. H. Hoover, E. H. Porter, and J. J. Zasio. SSIM: A software levelized compiled-code simulator. In *The Proceedings of the Design Automation Conference*, pages 2–8, June 1987.