Incremental Techniques for the Identification of Statically Sensitizable Critical Paths

Yun-Cheng Ju and *Resve A. Saleh*
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
Urbana, IL 61801

ABSTRACT

This paper describes new algorithms for finding the K-most critical paths, checking static sensitizability of these paths, and performing incremental timing verification on combinational circuits. The static sensitization method uses binary decision diagrams to avoid costly backtracking operation used in other path analysis programs. The speed and efficiency of the techniques are demonstrated individually using the ISCAS benchmark circuits, and then together in a timing optimization loop.

1. INTRODUCTION

Ensuring that a design meets a set of timing constraints is a very important issue in the design of integrated circuits. Circuit simulators such as SPICE are occasionally used for this purpose but they are usually too slow to verify an entire circuit. Timing analysis [JOU87, OUS85] is a data-independent approach which is typically much faster. Recent work [DU89, MCG89, BEN90] has improved this approach by identifying the so called *false paths* and only reporting the longest *true paths* in a circuit.

Even though timing analysis programs are relatively fast, it usually takes tens or hundreds of iterations of modification and verification to either correct all the timing violations or to optimize a design. However, when a design is slightly modified, only a small portion of the circuit characteristics will be affected compared with the previous analysis. In addition, it usually takes much less time to update invalid information than to do a complete analysis. If the analysis can be carefully decomposed into several independent phases, some phases can to be either totally skipped or easily updated to reduce the execution time and to give the designers faster feedback. In this paper, we exploit this property to perform incremental timing analysis on combinational circuits.

We begin by presenting an efficient scheme using *binary decision diagrams* to eliminate the *statically unsensitizable false paths* in a combinational circuit which is a key contribution in this paper. We then present an improved path enumeration algorithm to extract the K most critical paths that does not traverse any unnecessary full and/or partial paths. Finally, we present an *incremental timing verification* algorithm which can efficiently determine the longest statically sensitizable true paths if some of the gate delays are changed in the design. Experimental results are provided to illustrate that a large speed up can be achieved using our incremental algorithm. Since most of the *true paths*, and most *viable paths*, are *statically sensitizable*[MCG89] and since static sensitization is much cheaper to implement (especially in a timing optimization environment), circuit designers can use our incremental algorithm to correct all of the statically sensitizable long paths in a very short amount of time.

2. A STATIC PATH SENSITIZATION SCHEME

Whenever a long path in a circuit is found, it must be categorized as a true path or a false path. This section presents a new approach to the false path problem in timing verification.

Several sensitization conditions have been investigated in [MCG89]. The most primitive condition, i.e. *static sensitization* condition, is used in our work because, with the help of *binary decision diagrams* [BRY86, BRA90], accurate static sensitization can be very efficient, especially for incremental timing verification. This condition follows from the observation that for a signal to travel down a path $P = < f_0, f_1, \cdots f_m >$, changing the value of f_i must change the value of f_{i+1}. That is equivalent to saying that all of the *other inputs* to the gates on the critical path must have a *non-controlling stable* value (eg., 1 for AND gates and 0 for OR gates).

The basic idea of previous approaches reported [BEN90, DU89, PER89, MCG89] is illustrated in the example shown in Figure 1. They sensitize a critical path by generating and resolving logic constraints. Suppose the path $P_1 = <X, e, f, c, d>$ is the one of interest. In order to propagate X to e, Z must be 1 since e is an AND gate.

Propagating e to f produces no constraints, but propagating f to c requires $b = 1$ and propagating c to d requires $g = 0$. However, $g = 0$ implies both f and X are 1 and a conflict is detected at node e because $X = Z = f = 1$. The problem with previous approaches is that they have to compromise between accuracy and the combinatorial explosion associated with backward searches to justify these logic constraints. For example, if g is an AND gate instead, setting either f or X to 0 can temporarily satisfy the constraint but *backtracking* may be needed later to retrace the search graph to resolve any conflicts that may arise by trying alternative assignments at previously assigned nodes. In addition, the same partial path may have to be examined many times which makes the problem even worse.

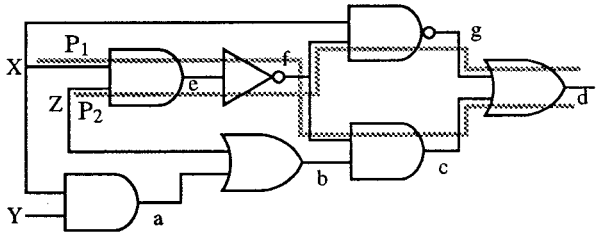


Figure 1: An Example for Static Sensitization

Our idea is based on the fact that, if the logic description of every node in terms of the primary inputs is available, no backtracking is necessary to sensitize a path. This backtracking operation is really the most expensive part of path sensitization, and it is completely eliminated in our approach. In other words, we transform the sensitization problem into a *satisfiability* problem and apply binary decision diagrams (BDDs) to facilitate the procedure. As we shall see in the next section, our direct approach is very efficient and accurate. Let's examine the path P_1 again. In order to sensitize path P_1, we need $Z = b = 1$ but $g = 0$. With the help of BDDs, it is very easy to see that $Z \cap b \cap g' = Z \cap (XY+Z) \cap (X'Z)' = \emptyset$. On the other hand, if we examine another path $P_2 = <Z, e, f, g, d>$, we need $X = 1$ and $c = 0$ to sensitize the path. Since $X \cap c' = X \cap (X'Z+XYZ')' = XY'+XZ \neq \emptyset$, path P_2 is statically sensitizable.

Since we are only interested in finding the longest true paths in a circuit, it is not necessary to examine all of the output functions at one time. As we will show in the next section, using binary decision diagrams (BDDs), the *logic functions* for all of the internal nodes of the slowest primary output function can be constructed in only a few CPU seconds for most of the ISCAS benchmark circuits.

2.1. Experimental Results

We implemented our static sensitization scheme using the BDD package described in [BRA90] and an ordering strategy described in [MAL88]. Figure 2 shows the execution times in CPU seconds for constructing the BDDs for the slowest primary output in each benchmark circuit and for sensitization while searching for the 3 most critical true paths. The path enumeration algorithm used will be explained in the next section. The table also shows the number of false paths detected by our scheme before reaching the third true path and the total number of nodes in the BDDs for each circuit. Circuits c432 and c2670 could be preprocessed and analyzed in under a minute even though there were a large number of false paths in them. The BDDs for circuit c6288 could not be generated (degenerate example of a multiplier) and circuit c1908 needed a depth first scheme to complete the search and had over 32,000 false paths! The other examples produced short analysis times.

Circuit	Time to build bdd	Size of bdd	Time to search	Total size	Nbr. of false paths
*c432	6.09	8022	48.41	21421	575
c499	25.97	13668	10.70	15548	0
c880	14.19	20865	1.63	23816	0
c1355	22.61	19477	10.27	23541	23
‡c1908	21.82	15466	39446.57	32179	32457
c2670	25.61	39709	33.52	46636	969
c3540	0.01	10	0.01	14	0
c5315	56.79	22743	5.18	25934	112
c6288	-	-	-	-	-
c7552	3.64	4431	1.12	4991	57

Time shown in CPU seconds on VAX3500 workstation
* original input ordering was used.
‡ depth first search algorithm was used.
- memory ran out before completion.

Figure 2: Sensitization Results

In addition, currently we are investigating the possibility of extending our scheme to handle the false path problem at switch level using the switch-level symbolic simulation techniques [BRY87].

3. A PATH ENUMERATION ALGORITHM

Several algorithms have been developed earlier [YEN89] to extract the K most critical paths from a combinational network; however, they may not be efficient enough to handle the cases when K is large. In this section we present an improved algorithm that enumerates the longest paths for a given acyclic directed graph in a non-increasing order of their delays. In the rest of this paper, we use *nodes* to represent components and use *edges* to represent the connections between components. To simplify the calculation of the delay of a path, we assume that the weight of a node is zero and the weight of each edge corresponds to the summation of the delay of component as well as the delay of connection. The efficiency of this algorithm is demonstrated by some experimental results.

The use of *branch slacks* is the key idea of our new algorithm. One significant advantage of our proposed algorithm is that circuit designer does not have to provide the constant K at the beginning of the analysis. Our algorithm generates the next longest path at every new iteration and can be stopped and/or continued at any time. There are two major phases in the algorithm : (1) the

computation of maximal delay to sink and ranking edges for each node, and (2) the path enumeration phase.

3.1. Preprocessing Phase

For the purposes of handling multiple primary inputs and primary outputs, we add a source node s and sink node t and add a dummy edge with weight zero from node s to each of the primary input nodes and from each of the primary output nodes to the node t. Without loss of generality, we assume each gate has equal rise and fall delays. For the case when rise and fall delays are different, a simple transformation proposed by [LI89] can be applied to represent the delay information. The delay D of a path $P = <s = v_0, v_1, v_2, ..., v_n, v_{n+1} = t>$ is defined as $D(P) = \sum_{i=0}^{i=n} d(v_i, v_{i+1})$, where the $d(v_i, v_{i+1})$ is the delay of edge $<v_i, v_{i+1}>$.

In order to determine the order of traversals of paths in the path enumeration phase, we use the basic ideas of calculating the *max_delay_to_sink*(v) for each node v and ordering the successors of each node v from [YEN89]. The *max_delay_to_sink*(v) is defined as the maximum of the delays of all possible partial paths starting from node v and ending with the sink.

While ordering the successors of each node v according to the non-increasing order of the cost function, $(d(v, u) + max\_delay\_to\_sink(u))$, the branch slack, *branch_slack*(v, u_i), can be computed as the difference of the cost functions between each adjacent successors u_i and u_{i+1}. Here, u_{i+1} is the node next to node u_i in the sorted successors list of v. Therefore, notationally, we can write that $u_{i+1} = nextnode(v, u_i)$. Note that *branch_slack* $(v, u_k) = d(v, u_k) + max\_delay\_to\_sink(u_k)$ if u_k is the last successor of node v. This branch slack is the key to the efficiency of the new algorithm described in the next section.

Figure 3 illustrates the calculation of *max_delay_to_sinks* and *branch slacks*. Suppose we are processing node C and that nodes D, F, and G have *max_delay_to_sink*'s of 10, 19, and 5, respectively. The path from C to D gives a maximum delay of $10 + 10 = 20$ whereas the path from C to G gives a maximum delay of only $12 + 5 = 17$. Therefore, D is the first successor of C, *max_delay_to_sink*$(C) = 20$, and *branch_slack*$(C, D) = 3$. Similarly, C is the first successor of B, *max_delay_to_sink*$(B) = 30$, and *branch_slack*$(B, C) = 30 - 29 = 1$.

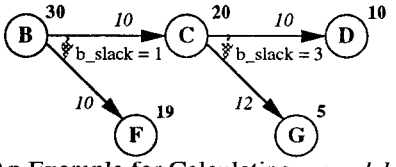
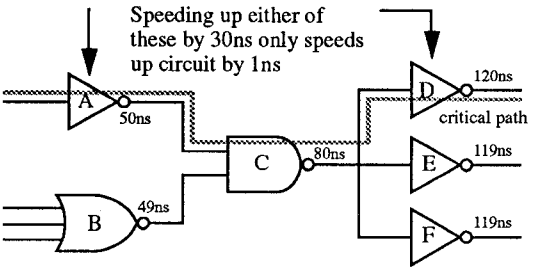
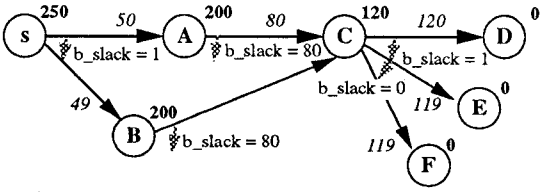


Figure 3: An Example for Calculating *max_delay_to_sink* and *branch_slack*

The branch_slack(u,v) gives two pieces of valuable information: (1) The delay of the next longest path which branches out from node u through another edge $<u, nextnode(u,v)>$ is *branch_slack*(u,v) shorter than the original path; and (2) The maximum gain in speed by reducing the delay associated with edge $<u,v>$ is *branch_slack*(u,v) since a new critical path may branch out before the improved edge $<u,v>$. Figure 4 demonstrates how branch slacks can solve the difficulties in speeding up designs mentioned in [JOU87]. Jouppi pointed out that even if the critical paths of a design are known, the best way to improve the performance of the design may not be clear. Figure 4(a) explains why speeding up some components does not improve performance much and figure 4(b) shows that branch slacks provide the exact information for circuit designers to avoid needless modifications.



(a) Difficulties in Speeding Up Designs



(b) Branch Slacks Identifies the Upper Limit for Performance Improvement

Figure 4: Branch Slacks Identifies the Best Spot to Improve a Design

3.2. The Path Enumeration Phase

The difference between our new algorithm and the previous ones is that our proposed algorithm can find the next longest path very easily with the help of branch slacks. Since all of the successors are sorted, the longest path $P_1 = <s = v_0, v_1, v_2, ..., v_n, v_{n+1} = t>$ can be found by expanding the longest partial path simply by adding the first successor of the last node on the partial path one at a time. With the help of the branch slacks associated with edges on P_1, it is not difficult to see that the second longest path P_2 branches out from node v_j on the path which has the smallest *branch_slack*(v_j, v_{j+1}) and the remaining nodes of path P_2 can also be determined by the same way as P_1 is generated. In general, at iteration k, the k longest paths are collected in the buffer and some of them may still have unexploited branch points. With the help of the smallest branch slack of all available branch points, the delay of the next critical path (*next_delay*) from which each path in the buffer can generate can be determined. Our algorithm simply picks the one with the longest

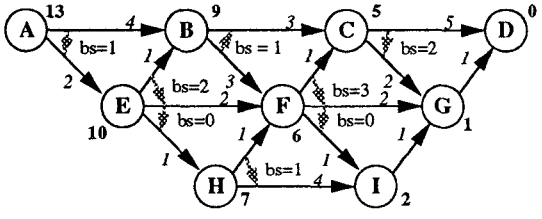
next_delay from a heap data structure to generate the k+1-th longest path. It should be noted that the use of a heap data structure is very important maintaining efficiency in critical path analysis programs [MCG89].

The overall path enumeration algorithm is outlined below. Notice that when a new path P_i is generated, all of its available branch points, v_j, and their associated *branch_slacks* are placed in *list*[i] according to the non-increasing order of their *branch_slacks*. In addition, the delay of the next critical path from which path P_i can generate, *next_delay*(P_1), is simply *delay*(P_1) minus the *branch_slack* of the first pair in *list*[i].

Algorithm : Path Enumeration

generate the longest path P_1;
prepare *list*[1] and calculate *next_delay*(P_1);
$k \leftarrow 1$;
while (k paths are not enough yet) {
  $i \leftarrow$ the path with the longest *next_delay*;
  $j \leftarrow$ the first available branch point in *list*[i];
  generate the next longest path P_{k+1} by branching out from the j-th node on path P_i;
  prepare *list*[k+1] and calculate *next_delay*(P_{k+1});
  remove the first element in *list*[i] and update *next_delay*(P_i);
  $k \leftarrow k + 1$;
}
return ($P_1 \cdots P_k$);

In order not to report the same path twice, when a new path P_k is generated by branching out from path P_i at node v_j, it shall not consider the branch points in front of v_j that are shared by both P_i and P_j. Figure 5 illustrates how our algorithm extracts the 5 most critical paths from an example network shown in Figure 5(a). A trace of the execution of our algorithm is listed in Figure 5(b).



(a) A Network with Source A and Sink D

iteration	path(delay)	next_ delay	available branch points (branch slack)
1	ABFCD(13)	12	A(1),B(1),C(2),F(3)
2	AEBFCD(12)	11	B(1),E(2),C(2),F(3)
3	ABCD(12)	10	C(2)
4	ABFCGD(11)	0	
5	AEBCD(11)	9	D(2)

(b) A Trace of the Execution of our Algorithm

Figure 5: Example of Extracting the 5 Most Critical Paths

3.3. Experimental Results

We implemented our algorithm in the C language on a VAX3500 workstation. Benchmarks were taken from the ISCAS benchmark suite [BRG85]. Random delays from 200 to 209 time units were assigned to each gate. Six large circuits (c1355, c1908, c2670, c5315, c6288, and c7552) with a total number of paths greater than 100,000 were tested using our algorithm. To provide a fair comparison, we improved the existing *best first* and *depth first* algorithms to extract K paths reported in [YEN89] so that they also used a heap. Figure 6 shows the execution times in CPU seconds and the percentages of wasted edge traversals, i.e., the percentages of edges encountered when generating paths which were eliminated later, for the three largest circuits. Notice that our algorithm does not make any unnecessary edge traversals and has the capability of continuing a search for the next most critical path without having to restart from scratch. Actually, the critical path analysis and path sensitization are done simultaneously in our algorithm and, in fact, the algorithm is expected to perform even better (relative to depth-first and best-first) when searching for the true paths because expanding each edge involves sensitization and is much more expensive.

Circuit	Algorithm	K=50	K=500	K=5000
c5315	Best First	0.10 (3%)	1.19 (3%)	40.24 (3%)
	Depth First	0.18 (58%)	0.91 (61%)	16.84 (64%)
	Ours	0.06 (0%)	0.52 (0%)	5.15 (0%)
c6288	Best First	0.22 (4%)	2.46 (5%)	61.30 (6%)
	Depth First	0.64 (70%)	8.06 (78%)	94.83 (83%)
	Ours	0.17 (0%)	1.49 (0%)	14.52 (0%)
c7552	Best First	0.08 (4%)	1.00 (3%)	32.77 (3%)
	Depth First	0.13 (70%)	1.15 (56%)	11.56 (59%)
	Ours	0.03 (0%)	0.44 (0%)	4.44 (0%)

Time Shown in CPU seconds on VAX3500 workstation

Figure 6: Performance Statistics

4. INCREMENTAL CRITICAL PATH ANALYSIS

In this section, we describe an incremental path analysis approach using the path analysis and sensitization techniques from the previous sections. A circuit design is usually analyzed for timing and modified tens or even hundreds of times during debugging and optimization cycles. Thus, it is very important for each iteration of the analysis loop to be as fast as possible. Incremental timing verification exploits the fact that when a design is slightly modified, only a few paths have to be re-examined.

In this paper, we focus on changes in gate delay values but not to the structure of a given circuit. Since timing information is not stored in the BDD-based logic representations, it is clear that there is no need to reconstruct the BDDs if only the delays of some components change. In addition, a full and/or partial path remains true (or false) if and only if it was a true (or false) path in the previous analysis. Therefore, we do NOT have to resensitize the long paths we examined in the previous runs and this is the main reason why we use static sensitization condition, even though it may underestimate the critical path delay for some rare cases. We believe it would be a

good idea to correct all of the statically sensitizable long paths first in a very short amount of time and then apply some more sophisticated timing verifiers to examine if there are any dynamically sensitizable or viable long paths left.

In general, the only characteristics that could change are the *max_delay_to_sinks*, *branch_slacks*, orderings of successors, and the *lengths* and *next_lengths* of existing partial/full paths which are all relatively cheap to update (< 1% CPU time) compared with the cost of building the BDDs and/or sensitizing paths. Instead of restarting the search from the beginning again, we only have to update the invalid information, reorder the heap, and continue the modified best first search of other long paths. In addition, since true paths are the only paths that need to be shortened while the lengths of false paths are not of interest, the lengths of true paths become shorter and shorter as the optimization phase proceeds, but the number of long false paths that other non-incremental algorithms have to sensitize becomes larger and larger. Our algorithm does not waste time re-examining known false paths.

The incremental search algorithm uses two heaps (the original path enumeration algorithm only uses one heap to keep the *next_delays* for every path). Suppose a circuit designer is interested in knowing the K most critical true paths. After updating the lengths of all of the paths in the buffer, the longest K most critical true paths (the buffer may have more than K true paths already) are put in the *Heap_L* with the shortest one on the top of the heap. On the other hand, the *next_lengths* of all of the paths with available branch points are put in another heap *Heap_N* with the longest one on the top of the heap. The overall incremental timing verification algorithm is outlined below:

Algorithm : Incremental Critical Path Identification

update the invalid information;
prepare *Heap_L* and *Heap_N*;
while (*Heap_N.top* > *Heap_L.top*) {
    /* *more unexploited long paths* */
    generate and sensitize a new long path P_{new};
    if (P_{new} is a true path)
        Replace the top element of *Heap_L* by P_{new}
            and rebuild *Heap_L*;
    Insert *next_delay* (P_{new}) in *Heap_N*;
    }
return (true paths in *Heap_L*);

The algorithm above only considers gate delay changes. The problem becomes somewhat more complicated if the circuit topology is changed. For example, the transduction method [MUR89, MAT89] can be used to eliminate redundant logic circuitry. In this case, the table of BDD logic functions can be updated in an event-driven manner but more paths have to be re-examined. However, we did not focus on this part because the CPU time spent on modifying the circuit topology usually dominates the entire timing optimization loop. This will be a topic of future consideration.

4.1. Experimental Results

In order to demonstrate the capabilities of the new algorithm, we designed an circuit timing optimization loop using our incremental timing verifier. At every iteration the most critical true path is identified and the delay of slowest device on the path is reduced by 5%. This loop is repeated until the modified design is 5% faster than the original design.

Two benchmark circuits, c1355 and c499, were chosen for this experiment because, as shown in Figure 2, these two circuits have only a few long false paths and they actually spend more time in sensitizing their paths than constructing the BDD tables Therefore, lower speed-ups are expected for these cases. Note that our algorithm neither re-examines the known long false paths nor re-constructs the BDDs after the first iteration. Figure 7 shows the accumulated CPU times (in seconds) for the whole optimization processes as a function of the iteration count. For both benchmark circuits, if no incremental technique is applied, the accumulated CPU time (labeled by Orig.) grows very quickly. However, if we exploit the fact that the BDD tables do not have to be re-constructed, a significant amount of CPU time can be saved (labeled by Incr_0). Finally, if our incremental search scheme is also used, the accumulated CPU time hardly increases after a few iterations (labeled by Incr.).

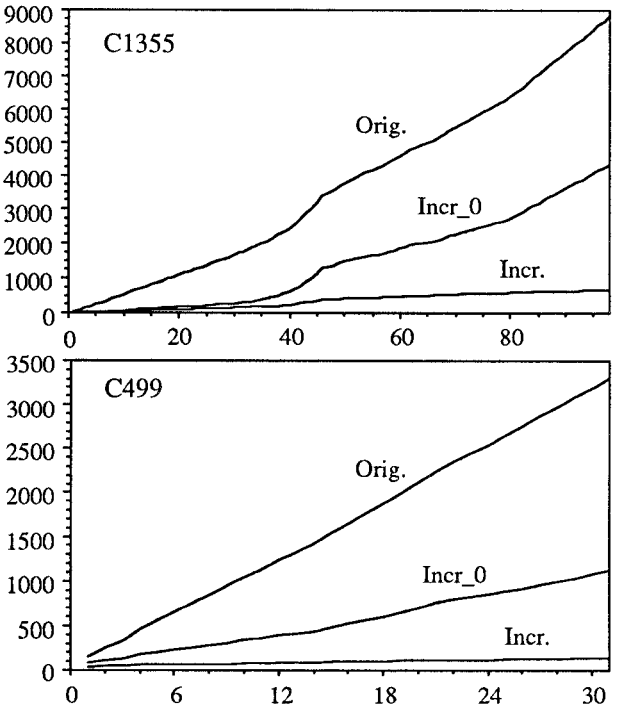


Figure 7: Accumulated CPU times (secs)

The average CPU time taken in each iteration and the maximum memory used for both algorithms are also shown in figure 8. With some fair amount of additional memory (20% for c499 and 58% for c1355), our incremental schemes drastically reduced the amount of CPU time spent on the timing optimization loop from 2.5 CPU

hours down to 11 CPU minutes for c1355 and from 1 CPU hour down to 2 CPU minutes for c499. As mentioned above, these were the particularly difficult circuits. For the circuits where all of the long paths have been sensitized earlier, our algorithm is thousands of times faster.

Circuit		c1355	c499
Number of Iterations		98	31
Average CPU seconds per iteration	Original	89.97	106.77
	Incremental	6.81	4.33
Speed Up	Overall	13.21	24.66
	Maximum	614.9	5276.4
Maximum Memory Usage	Original	60314	18648
	Incremental	95148	22458
	Overhead	58%	20%

Figure 8: Performance for Incremental Timing Verifier

5. CONCLUSION

We have presented new algorithms for path enumeration, static sensitization, and incremental timing verification for critical path analysis of combinational circuits. We demonstrated that all of these three algorithms are very useful for the application of timing optimization using the ISCAS benchmark circuits. The next step of this research is to apply the switch-level symbolic simulation techniques to extend the false path problem down to switch level and to apply the incremental approach when the structural changes are made to the circuit.

6. ACKNOWLEDGEMENTS

The authors would like to thank Karl. S. Brace and Randal E. Bryant for providing the BDD package, and Larry Jones and K. C. Chen for useful discussions. Financial support for this research was provided by the Semiconductor Research Corporation.

REFERENCES

[BEN90] J. Benkoski, E. V. Meersch, L. J. Claesen, and H. D. Man, "Timing Verification Using Statically Sensitizable Paths", *IEEE Transactions on Computer-Aided Design*, CAD-9 (no.10), pp. 1073-1083, Oct. 1990.

[BRA90] K. S. Brace, R. L. Rudell, and R. E. Bryant, "Efficient Implementation of a BDD Package", in *Proceedings of 27th ACM/IEEE Design Automation Conference*, pp. 40-45, July 1990.

[BRG85] F. Brglez and H. Fujiwara, "A neutral netlist of 10 combinational benchmark circuits and a target translator in fortran", *IEEE International Symposium on Circuits and Systems*, June 1985.

[BRY86] R. E. Bryant, "Graph-based Algorithms for Boolean Function Manipulation", in *IEEE Transactions on Computer*, C-35 (no.8), pp. 667-691, June 1986.

[BRY87] R. E. Bryant, "Boolean Analysis of MOS Circuits", in *IEEE Transactions on Computer-Aided Design*, CAD-6 (no.4), pp. 634-649, July 1987.

[DU89] D. H. Du, S. H. Yen, and S. Ghanta, "On the General False Path Problem in Timing Analysis", in *Proceedings of 26th ACM/IEEE Design Automation Conference*, pp. 555-560, July 1989.

[JOU87] N. Jouppi, "Timing Analysis and Performance Improvement of MOS VLSI Designs", *IEEE Transactions on Computer-Aided Design*, CAD-6 (no.4), pp. 650-665, July 1987.

[LI89] W. Li, S. Reddy, and S. K. Sahni, "On Path Selection in Combinational Logic Circuits", *IEEE Transactions on Computer-Aided Design*, CAD-8 (no.1), pp. 56-63 Jan. 1989

[MAL88] S. Malik, A. R. Wang, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Logic Verification using Binary Decision Diagrams in a Logic Synthesis Environment", in *Proceedings of IEEE International Conference on Computer Aided Design*, pp. 6-9, Nov. 1988.

[MAT89] Y. Matsunaga and M. Fujita, "Multi-Level Logic Optimization Using Binary Decision Diagrams", in *Proceedings of IEEE International Conference on Computer Aided Design*, pp. 556-559, Nov. 1989.

[MCG89] P. McGeer and R. K. Brayton, "Efficient Algorithms for Computing the Longest Viable Path in a Combinational Network", in *Proceedings of 26th ACM/IEEE Design Automation Conference*, pp. 561-567, July 1989.

[MUR89] S. Muroga, Y. Kambayashi, H. C. Lai, and J. N. Culliney, "The Transduction Method - Design of Logic Networks Based on Permissible Functions", *IEEE Transactions on Computer-Aided Design*, CAD-38 (no.10), pp. 1404-1424 Oct. 1989

[PER89] S. Perremans, L. Claesen, and H. D. Man, "Static Timing Analysis of Dynamically Sensitizable Paths", in *Proceedings of 26th ACM/IEEE Design Automation Conference*, pp. 568-573, July 1989.

[OUS85] J. Ousterhout, "A Switch-Level Timing Verifier for Digital MOS VLSI", *IEEE Transactions on Computer-Aided Design*, CAD-4 (no.3), pp. 336-349, July 1985.

[YEN89] S. H. Yen, D. H. Du, and S. Ghanta, "Efficient Algorithms for Extracting the K Most Critical Paths in Timing Analysis", in *Proceedings of 26th ACM/IEEE Design Automation Conference*, pp. 649-654, July 1989.