Identification of Viable Paths Using Binary Decision Diagrams

Yun-Cheng Ju and Resve A. Saleh

Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
Urbana, IL 61801

Abstract
This paper describes an efficient algorithm for the identification of viable paths in a combinational logic circuit using binary decision diagrams. The viable paths are justified by generating and resolving logic and delay constraints along the critical path based on the stable times and stable values of the side inputs. Results of the analysis, using the ISCAS combinational benchmark circuits, indicate that most of the circuits can be analyzed in a few CPU-seconds.

1. Introduction
Timing analysis is a data-independent approach to verify whether a design meets a set of timing constraints. Programs like Crystal [1] and TV [2] analyze the circuit and compute the delays along each path in the network. They are used either alone to estimate a critical path delay, or to identify the critical path for later analysis by a circuit simulator, which is typically much slower. However, not all of the long paths are of interest to the designers. In fact, the main path of interest is the one with the longest delay down which signals can actually propagate. Recent work [3, 4, 5] has improved this approach by identifying the so called false paths and only reporting the longest true paths in a circuit.

The false path problem follows from the observation that, for a signal to propagate down a path \( P = < f_0, f_1, f_2, \ldots, f_n, f_{n+1}, \ldots > \), changing the value of \( f_i \) must change the value of \( f_{i+1} \). In the rest of this paper, the terms node and gate are used interchangeably. An event is defined as a transition of logic value at node \( f_i \) along the critical path either from 1 to 0 or from 0 to 1 and the arrival time of this event is merely the sum of the circuit delays along the partial path terminating in \( f_i \), a number we shall call \( \tau_i \). To be more specific, \( \tau_i = \sum_k d_G(f_k) \) where \( d_G(f_k) \) is the gate delay associated with gate \( f_k \).

A sensitization condition defines what constitutes a true path. Several sensitization conditions have been investigated in [4, 5]. The limitations of static and dynamic sensitization condition were examined in [5] and a more robust sensitization condition called the viability condition was presented. It ignores the side inputs with path delays greater than or equal to the corresponding delay of the critical path at a given point. A dynamic programming procedure for viability was also proposed in [5]. However, the performance was not satisfactory because only a few small circuits could be handled and no performance statistics were provided. Based on a new path enumeration algorithm [6], an efficient and accurate scheme to identify whether a path satisfies viability by generating and resolving logic and delay constraints along the critical path using binary decision diagrams (BDDs) [7] is described in this paper.

2. Viability Condition
In order to be conservative, the viability condition ignores the stable values for late side inputs. Consider the example shown in Figure 1. Let the arrival time of \( c \) be \( \tau_c = 8 \) under some input vectors and the maximum stable times for partial paths terminating at the side inputs \( a \) and \( b \) be 10 and 7, respectively, under the same set of input vectors. Then, the viability condition requires node \( b \) to have a non-controlling stable value 1 but does not impose any restriction on the stable value of \( a \) because it is a late side input. Now, reconsider the example in the context of the delay estimation accuracies of node \( a \) and \( b \) and its impact on path viability. If the stable time for a side input is underestimated (eg., if the maximum stable time of \( a \) is miscalculated to be 6), then we may introduce some unnecessary constraints and the result may be wrong because we may lose viable paths. On the other hand, if the stable time for a side input is overestimated (eg., if the maximum stable time of \( b \) is miscalculated to be 10), then we may not consider the path viable and the result may also be wrong because we may miss viable paths.

![Figure 1: Viability Sensitization Condition](image-url)
be 9), then we may miss some necessary constraints and the result becomes less accurate because some false paths may be incorrectly identified as a viable path.

Figure 2 further illustrates the problem described above. Suppose all gates have a fixed delay of 1 and we are considering the path \( P_1 = \langle b, x, z, \text{out} \rangle \). Initially, node \( y \) has a maximum stable time of 2. After sensitizing the path from \( b \) to \( z \), we know that node \( a \) must be 0 in order to propagate \( x \) to \( z \) through an OR gate. Therefore, the stable time for \( y \) now reduces to 1. Previously, we did not need a constraint for \( y \) since it was considered a late side input, but now it must hold a non-controlling value because we now know it is an early side input. In order to handle cases like this, every side input considered to be late in an earlier pass should be rejustified when the set of input vectors that sensitize the critical path changes.

3. New Viability Algorithm

We propose a new scheme using BDDs to automatically justify late side inputs by generating and resolving constraints. Similar to the approach proposed in [6], the logic function of each required node is determined and stored as a BDD. In addition, the maximum and minimum stable times of each node \( v \) of the circuit with the worst case delay configuration are also calculated from the source node to the sink node.

Our new two-phase algorithm for sensitizing a viable path proceeds as follows. Initially, the sensitizing set \( S \) is set to 1 implying that no restrictions are placed on the input vectors at the beginning. During the first phase, we traverse the long path under consideration from the source node to the sink node and sum up the gate delays along the path to produce the arrival time \( \tau_i \) at the output of each gate \( f_i \). If any side input, say node \( v \), has a maximum stable time \( T_{\text{stable max}}(v) \leq \tau_i \), then it is an early side input and must hold a non-controlling stable value. In this case, the sensitizing set \( S \) is updated by ANDing it with \( v \) or \( v' \), depending on the gate type. The maximum and minimum stable times of the direct and transitive fanouts of node \( v \) are also updated. If \( T_{\text{stable max}}(v) \geq \tau_i \), it is considered as a late side input for the moment and the logic constraint, "If \( T_{\text{stable max}}(v) < \tau_i \) then \( v \) should hold a non-controlling stable value", is generated at the node for later use.

If the sensitizing set \( S \) becomes empty, i.e., \( S = \emptyset \), after phase one, then the path can not be viable and we stop. Otherwise we enter phase two and double check all of the constraints generated in phase one. We first check to see if \( S \) can satisfy all constraints without considering their stable times. If the answer is true then this path is not only viable but also statically sensitizable. However, if the path fails to satisfy all the non-controlling values, our constraint generating and resolving scheme, to be described next, will be used to check for possible inconsistency among those constraints. Unless we run out of memory in manipulating BDDs, our new algorithm always terminates since the range of possible stable times for every node is bounded and monotonically reducing. The outline of the algorithm is listed below:

**Algorithm: Path Viability Analysis**

```plaintext
/* Phase One : Examine early side inputs for path */
P = \langle f_0, f_1, f_2, \ldots, f_n \rangle /*
\begin{align*}
S & \leftarrow 1; \\
i & \leftarrow 1;
\end{align*}
while (i <= n) {
    for (all early side input v of fi) 
        update S and stable times.
    for (all late side input v of fi) 
        generate a logic constraint for v.
    if (S = \emptyset) then return ("false path");
}
/* Phase Two : Double check the late side inputs */
if (S \cap all non-controlling values \neq \emptyset) 
    then return (statically sensitizable path )
while (there is any untested constraint left) {
    pick the next untested constraint; check for this constraint and update S;
    if (S = \emptyset) then return (false path); 
    generate new constraints and/or drop old constraints if necessary;
    if (S or some T_{\text{stable max}}, T_{\text{stable min}} changes) 
        then set all constraints untested;
} return (possible viable path);
```

4. Constraint Generation and Resolution Scheme

Two types of constraints are used in our scheme: one is referred to as a logic constraint and the other is referred to as a delay constraint. A logic constraint requires a side input to hold a specific stable value (either 0 or 1) if the maximum stable time associated with that particular node drops below a specific threshold \( \tau \), i.e., if the particular node becomes an early side input. On the other hand, a delay constraint requires a side input to be a late side input, i.e., the maximum stable time associated with that particular node must be at least as long as the threshold \( \tau \).

Suppose the worst case gate delay of a gate \( v \) is \( d_G(v) \) and gate \( v \) has \( n \) input signals \( i_1, i_2, \ldots, i_n \). Then,
the logic and delay constraints are handled as follows.†

4.1. Logic Constraints

Given a gate \( v \), a stable value \( SV \), and a threshold time \( \tau \), the logic constraint "If \( T_{\text{stable-max}}(v) < \tau \) then \( v = SV \)" requires gate \( v \) to have a stable value \( SV \) if the maximum stable time \( T_{\text{stable-max}}(v) \) with respect to the sensitizing set \( S \) of the critical path drops below \( \tau \), i.e., when \( v \) becomes an early side input. Without loss of generality, we assume that the desired \( SV = 1 \). Four steps, depending on the stable time of \( v \) and whether \( v \) has been assigned a stable value, are taken to examine whether it can be satisfied or not.

(L1) If \( T_{\text{stable-min}}(v) \geq \tau \) (\( v \) is now a permanent late side input) or \( S \subseteq v \) (\( v \) always evaluates 1 under input vectors \( S \)), then the constraint can always be satisfied. Drop this redundant constraint.

(L2) If \( T_{\text{stable-max}}(v) < \tau \) (\( v \) is an early side input), then assign \( v \leftarrow 1 \) and set \( S \leftarrow S \cap v \). In addition, update the maximum and minimum stable times of the direct and transitive fanouts of \( v \).

(L3) If \( S \subseteq v \), then \( v \) will always evaluate to 0 so that in order to satisfy the viability constraint, the stable time must be greater than or equal to \( \tau \). Replace the constraint with a delay constraint "\( T_{\text{stable-max}}(v) \geq \tau \)."

(L4) Backward Constraint Propagation

When all of the above three tests fail, a case study is needed to see whether the constraint can be transformed to other constraints or be propagated to the inputs of gate \( v \).

(a) if \( v \) is an AND gate

Decompose it into \( n \) new constraints (one for each input to gate \( v \)) as follows: "If \( T_{\text{stable-max}}(i_j) < \tau - d_G(v) \) then \( i_j = 1 \), where \( j = 1, \ldots, n \). Since 0 is the controlling value of AND gates, any input \( i_j \) has a stable value 0 and a stable time \( T_{\text{stable-max}}(i_j) < \tau - d_G(v) \), then a conflict is detected and the path is not viable.

(b) if \( v \) is an OR gate

Fewer actions can be taken in this case because 1 is the controlling value of OR gates. Even though some input node \( i_j \) may have a short stable time, it does not need to have a stable value 1 unless all of the other inputs are early and have a non-controlling stable value 0. When this special case occurs, i.e., \( T_{\text{stable-max}}(i_j) < \tau - d_G(v) \) and \( S \subseteq i_j' \) for all \( j \neq k \), replace it with a new constraint, "If \( T_{\text{stable-max}}(i_k) < \tau - d_G(v) \) then \( i_k = 1 \)." Of course, if all inputs have the same stable values of 0 and stable time \( T_{\text{stable-max}}(i_j) < \tau - d_G(v) \), then the path is not viable.

4.2. Delay Constraints

Given a gate \( v \), and a threshold time \( \tau \), a delay constraint requires gate \( v \) to have a maximum stable time \( T_{\text{stable-max}} \) at least \( \tau \) long, i.e., \( T_{\text{stable-max}}(v) \geq \tau \). In other words, \( v \) must be a late side input. It is stronger than a logic constraint because having a longer stable time automatically satisfies the logic constraint. A three-step procedure, depending on the stable time of \( v \), is taken to examine whether there is any conflict.

(D1) If \( T_{\text{stable-max}}(v) \geq \tau \) (\( v \) is a late side input), then the constraint is redundant. Drop this constraint.

(D2) If \( T_{\text{stable-max}}(v) < \tau \) (\( v \) is an early side input), then a conflict is found and this path cannot be viable. Set \( S \leftarrow \varnothing \).

(D3) Backward Constraint Propagation

Again, a case study is needed if both of the tests above fail.

(a) if \( v \) is an AND gate

if all inputs \( i_j \)'s except \( i_k \) have an early stable time \( T_{\text{stable-max}}(i_j) < \tau - d_G(v) \) and a known non-controlling stable value 1 (\( S \subseteq i_j \)), replace it with a new constraint; "If \( T_{\text{stable-max}}(i_j) \geq \tau - d_G(v) \). If the first test fails, we examine each input of gate \( v \) for its stable value. Since 0 is the controlling value of AND gates, if any input \( i_j \) has a stable value 0, i.e., \( S \subseteq i_j \) and \( S \subseteq i_j' \), then the stable time for node \( v \) is at most \( \tau_i + d_G(v) \). The key point here is that we need either \( i_j = 1 \), or \( T_{\text{stable-max}}(i_j) \geq \tau - d_G(v) \). If \( i_j = 0 \) (\( S \subseteq i_j' \)), generate a new delay constraint, "If \( T_{\text{stable-max}}(i_j) \geq \tau - d_G(v) \). On the other hand, if the stable value is unknown, i.e., neither \( S \subseteq i_j \) nor \( S \subseteq i_j' \), replace it with a new logic constraint: "If \( T_{\text{stable-max}}(i_j) < \tau - d_G(v) \) then \( i_j = 1 \)."

(b) if \( v \) is an OR gate

Similar to the previous case except replacing \( i_j' \) with \( i_k' \) and \( i_k' \) with 0's.

4.3. Examples

The basic ideas are illustrated by two examples using the circuit shown in Figure 2. The update operation of maximum and minimum stable times described in phase one is bypassed to better illustrate our algorithm. Suppose we want to examine the viability of path \( P_1 = \langle b, x, z, \text{out} \rangle \). When traversing \( x \rightarrow z \), a logic constraint, "If \( T_{\text{stable-max}}(c) < 0 \) then \( c = 1 \), is generated. Certainly, this one is dropped by (L1) later because \( T_{\text{stable-max}}(c) = 0 \). When traversing \( x \rightarrow z \), both \( a \) and \( b \) are set to 0, resulting in \( S = a'b' \) because \( a \) and \( b \) are early side inputs. Finally, when traversing
z \rightarrow \text{out}, another logic constraint, "\text{if} \ T_{\text{stable, max}}(y) < 2 \) then \ y = 1", is generated. This new constraint is transformed into a delay constraint, "\text{if} \ T_{\text{stable, max}}(y) \geq 2", due to (L3) since \( S \subseteq y' (S = a'b', y = a(b+c)) \). Later, this new constraint is decomposed into two new constraints, "\text{if} \ T_{\text{stable, max}}(a) \geq 1" and "\text{if} \ T_{\text{stable, max}}(w) < 1) then \ w = 1", from (D3a) because \( S \subseteq a' \) but the stable value of \( w \) is still unknown. Since the first one causes a conflict (\( T_{\text{stable, max}}(a) = 0 \)), this path is not viable. In fact, our algorithm notices that the maximum stable time of node \( y, T_{\text{stable, max}}(y) \), reduces to 1 after traversing \( x \rightarrow z \) because \( a \) is set to 0. A conflict is detected immediately when traversing \( z \rightarrow \text{out} \ (S \cap y = a'b' \cap a(b+c) = \emptyset) \) because node \( y \) is no longer a late side input.

For the second path \( P_2 = <c, w, y, \text{out}> \), when traversing \( c \rightarrow w \), a logic constraint, "\text{if} \ T_{\text{stable, max}}(b) < 0 \) then \ b = 0", is generated. When traversing \( w \rightarrow y \), \( a \) is set to 1 because \( a \) is an early side input, resulting in \( S = a \). Finally, when traversing \( y \rightarrow \text{out} \), another logic constraint, "\text{if} \ T_{\text{stable, max}}(z) < 2 \) then \ z = 1", is generated. However, since \( z = a + b + bc = a + b \) and \( S \cap z = b' \neq \emptyset \), this path is a statically sensitizable viable path.

5. Results and Conclusions

We implemented our proposed scheme in the C language on a DEC-5000 workstation using the BDD package developed by [8] and an ordering strategy developed by [9]. Benchmarks were taken from the ISCAS benchmark suite. Random delays from 200 to 209 time units were assigned to each gate. Since we are only interested in the critical paths, it is not necessary to build the BDDs for all primary output functions in a circuit. Figure 3 shows the execution time in CPU seconds and the total BDD size in finding the longest three viable paths for the slowest functions of ISCAS benchmark suite. Our algorithm required a few CPU seconds to analyze each circuit. Notice that no performance statistics were provided in [5]. In addition, we were able to find the longest viable paths for benchmark circuit c1355, c2670, and c5315 which had not been done previously. We also noticed that the longest viable paths are also statically sensitizable in most of these circuits.

To summarize, a new algorithm for the identification of viable paths in combinational logic circuits has been presented. The algorithm is very effective as long as the BDDs for the circuit nodes can be easily generated.

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 was provided by the Semiconductor Research Corporation.

References