Computation of Floating Mode Delay in Combinational Circuits: Practice and Implementation

Srinivas Devadas	Kurt Keutzer	Sharad Malik	Albert Wang
Department of EECS	Synopsys	Department of EE	Synopsys
MIT, Cambridge	Mountain View	Princeton University	Mountain View

Final Manuscript

Abstract

Delay computation in combinational logic circuits is complicated by the existence of unsensitizable (false) paths and this problem is arising with increasing frequency in circuits produced by high-level synthesis procedures. Various sensitization conditions have been proposed in the past to eliminate false paths in logic circuits, but we use a recently developed *single-vector* condition, that is known to be necessary and sufficient for a path to be responsible for the delay of a circuit (*i.e.* true) in the floating delay model.

In this paper we build on this theory and develop an efficient and correct delay computation algorithm, for the floating mode delay. The algorithm uses a technique we call *timed-test generation* and can be incorporated into any stuck-at fault test generation framework. We describe in detail an implementation of the timed-test generation algorithm that uses both logical and timed, forward/backward implication and backtrace procedures to simultaneously prove the truth or falsity of sets of paths in the circuit. Logical and temporal conflict detection during implication and backtrace are used to speed up the algorithm. Unlike previous techniques, the algorithm remains highly efficient even when a large number of distinct gate and path delays exist in the given circuit.

We provide a comprehensive set of results that show significant speed-ups over previous delay computation techniques.

1 Introduction

Delay computation in combinational logic circuits is complicated by the existence of unsensitizable (false) paths and this problem is arising with increasing frequency in circuits produced by high-level synthesis procedures. Various sufficiency conditions for a path to be responsible for the delay of the circuit have been offered in the past. The general framework for this work has been the observation that for a path to be true in the combinational circuit under consideration it must be true in the floating delay model [4]. The floating delay model is a model for the delay of a circuit that looks at the sensitization of paths in the circuit on a single vector and makes conservative assumptions about the prior state of the circuit on application of that vector. Work in this model has included results by [2] [1] [9] and [4].

The two important issues in eliminating false paths are correctness and efficiency. While the work of [2, 9] and others made significant strides in the direction of correct false path identification even the recent work of [9], that proposed a condition called *viability*, could not be said to correctly identify the paths that were responsible for the delay of the circuit, although it could accurately identify the delay of the circuit in the floating delay model.

The ability to correctly identify paths that are responsible for the delay of a circuit is absolutely necessary for proper performance optimization, but it was not until the work of [4] that a condition was offered that correctly identified true paths in the floating delay model. An equivalent necessary and sufficient condition for a path to be true in the floating delay model was given with an independent derivation in [5] and there the failure of the viability criterion was also demonstrated.

Given necessary and sufficient conditions for a path to be true the problem remains to eliminate false paths and determine the correct delay of a circuit in a computationally efficient manner. Correctness issues aside, the principal problem with previous false-path eliminating static timing verifiers has been that they compute the truth or falsity of one path at a time. This approach can be unacceptably slow for circuits with large numbers of paths. The work of [5] introduced a general *timed-test generation* strategy that enabled the consideration of entire sets of paths, rather than individual paths, avoiding the need for explicit path enumeration in false path identification. While the results presented for a simple implementation of timed test generation in [5] resulted in substantial speed improvements over the implementations of [4, 9], the initial implementation still left much to be desired in terms of efficiency. In this paper, we use the general timed-test generation framework of [5] but provide details on the implementation techniques necessary to make a computationally efficient floating delay calculator. To use stuck-at-fault testing methods, in [5] the analogs of the stuck-at-fault and the D calculus were introduced, but in this paper we show how a computationally efficient test generator can be constructed.

We provide a comprehensive set of results that show significant speed-ups over previous delay computation techniques.

This paper is organized as follows: In Section 2 we provide the notation and terminology used. In Section 3, we briefly review the work of [4, 5]. In Section 4, we present our global strategy for timed test generation, that uses timed forward and backward implication, and timed backtrace procedures. The timed forward implication, backward implication and backtrace procedures are described in Sections 5, 6 and 7, respectively. Experimental results on a wide variety of circuits are provided in Section 8.

2 Definitions and Notation

As this paper will span the areas of synthesis, testing and timing analysis we need to provide minimal amount of terminology from these fields.

2.1 Logic Synthesis

A **Boolean function** F of N variables is a mapping from $B^N = \{0,1\}^N \rightarrow \{0,1\}$. We model B^N as a Binary N-cube. A vertex $v \in B^N$ for which $F(v) = 1$ is a member of the ON-set. If $F(v) = 0$ then v is a member of the OFF-set.

A **literal** is a Boolean variable or its complement. We formally define a **cube** as a set of literals and we interpret the cube as a product of literals. For example $\{a, b, c\}$ is a cube, interpreted as $a \cdot b \cdot c$ which may be abbreviated to abc. A **minterm** is a cube in which every variable in the Boolean function appears. We may interpret the minterm as a **vertex** in the N-cube.

Minterms and cubes may be used to represent the values of a set of input variables: *e.g.* $x\overline{y}z$ is shorthand for $x = 1,\ y = 0$, and $z = 1$. In this way there is a natural correspondence between an input vector or input stimulus, a minterm and a vertex in the N-cube. This correspondence may be extended to cubes where unspecified values in the function are assumed to be undefined values. Thus, if a circuit C has inputs $v,\ w,\ x,\ y,\ z$ then applying the cube $x\overline{y}z$ to C is shorthand for applying $v = X,\ w = X,\ x = 1,\ y = 0$, and $z = 1$. (Here X denotes an unknown value.)

2.2 Testing

A Combinational Logic Circuit is represented as a labeled, directed, acyclic graph (DAG) $G = (V, E)$ with each vertex v labeled with the name of a primitive gate such as AND, OR or NOT, or with the name of a primary input or output. In general, a combinational logic circuit can be created from gates of arbitrary complexity, but to simplify the discussion we assume that the circuit is expressed in terms of primitive gates. There is an edge $< u, \ v >$ in V between two vertices if the output of the gate associated with u is an input to gate v. The members of V that have no fan-in are the only vertices that may be labeled with the name of a primary input. The members of V that have no fan-out are the only vertices that may be labeled with the name of a primary output.

If the output of a gate, g_1, is connected to an input of gate, g_2, g_1 is a **fanin** of g_2. Gate g_2 is a **fanout** of gate g_1.

A single-output combinational logic circuit computes a Boolean function in the obvious way. An N input M output combinational logic circuit computes a pseudo-Boolean function from $B^N \rightarrow B^M$. As we will be dealing primarily with combinational logic circuits in this paper, we will use circuit to mean combinational logic circuit.

A gate has an input/output **stuck-at-1** if the logical value associated with the input/output is 1 independently of the value presented at the input. **Stuck-at-0** can be defined similarly. A circuit has a **single-stuck-at-fault** if there is one stuck-at-fault in the circuit. A circuit has a **multiple-stuck-at-fault** (multifault) if there are one or more stuck-at-faults in the circuit.

For a fuller treatment of testing terminology, see [3, 7].

2.3 Timing

In this section we introduce terminology that will allow us to discuss timing issues as well as fault models for temporal behavior.

A **path** in a combinational circuit is an alternating sequence of vertices and edges, $\{v_0, e_0, ..., v_n, e_n, v_{n+1}\}$, where edge e_i, $1 \leq i \leq n$, connects the output of vertex v_i to an input of vertex v_{i+1}. For $1 \leq i \leq n$, v_i is a gate; v_0 is a primary input and v_{n+1} is a primary output. Each e_i is a net. With each vertex v (edge e) we associate a delay $d(v)$ $(d(e))$.

The **length** of a path $P = \{e_0, v_0, e_1, ..., e_n, v_n, e_{n+1}\}$ is defined as $D(P) = \sum_{i=0}^{n} d(v_i) + \sum_{i=0}^{n+1} d(e_i)$.

An **event** is a transition $0 \rightarrow 1$ or $1 \rightarrow 0$ at a gate. Consider a sequence of events, $\{r_0, \ r_1, \ ..., \ r_n\}$

occurring at gates $\{g_0,\ g_1,\ ...,\ g_n\}$ along a path, such that r_i occurs as a result of event r_{i-1}. The event r_0 is said to propagate along the path.

A **controlling value** at a gate input is the value that determines the value at the output of the gate independent of the other inputs. For example, 0 is a controlling value for an AND gate. A **non-controlling value** at a gate input is the value which is not a controlling value for the gate. For example, 1 is a non-controlling value for an AND gate. We say a gate g has a **controlled value** if one of its inputs has a controlling value; otherwise, we say g has a **non-controlled value**.

Let $\pi = \{v_0, e_0, ..., v_n, e_n, v_{n+1}\}$ be a path. The inputs of v_i other than e_{i-1} are referred to as the **side-inputs** to π. We say that π is **statically sensitizable** if there exists an input vector such that all the side-inputs along π settle to non-controlling values.

Let $\pi = \{v_0, e_0, ..., v_n, e_n, v_{n+1}\}$ be a path in a circuit C. We say that an input vector w **statically sensitizes to a 1(0)** path π in C iff the value of v_{n+1} is 1(0) and for each v_i, $1 \leq i \leq n+1$, if v_i has a controlled value then the edge e_{i-1} is the only input of v_i that presents a controlling value.

We say that an input vector w **statically co-sensitizes to a 1(0)** path π in C iff the value of v_{n+1} is 1(0) and for each v_i, $1 \leq i \leq n+1$, if v_i has a controlled value then the edge e_{i-1} presents a controlling value.

Note that if a path π is sensitized to a 1(0) then it is either statically sensitized to a 1(0) or it is statically co-sensitized to a 1(0). If a path is not sensitized to a 1 or 0 then the only other static behavior of a path on an input vector is that it may present a non-controlling value at a gate on the path and a path along some side-input presents a controlling value at that gate.

The **critical path** is the longest sensitizable path in the circuit. The delay of a circuit is the length of the critical path.

3 Review of Relevant Theory

In [4] a condition that is necessary and sufficient for a path to be responsible for the delay of a circuit, was presented. This condition is summarized below.

Proposition 3.1 *For a path π to be responsible for the delay of a circuit for input vector v, the following must be true at each gate g along the path:*

1. *If all inputs are non-controlling at g, then π must be the last to present the non-controlling value.*

2. *If there is even one controlling value at the inputs to g, then π must be the first to present the controlling value.*

The condition above directly translates to false path identification algorithms that operate on a *per path basis*. In [5] an alternative condition, equivalent to the one above, is given in terms of the detection of multifaults in an ENF description of a circuit or alternatively in a leaf-DAG. This condition naturally leads to a delay analysis procedure that avoids explicit path enumeration. In particular, determining the truth or falsity of a set of paths Π, all of length $\geq \delta$, was reduced to one of generating a test for a multiple stuck-at fault in the circuit, along paths in Π.

To avoid an explicit construction of a leaf-DAG or ENF representation of a circuit a timed D calculus was presented in [5] to allow for efficient multiple stuck-at fault test generation.

We briefly outline this procedure below: Assume a multilevel circuit C', where all inverters are at the primary inputs. For each edge e_i in C', a quantity $max(e_i)$ is determined which is the length of the longest path starting at the head of edge i and ending at the primary output of the circuit. In the multilevel circuit each fanout edge e_i of a primary input for which $max(e_i) \geq \delta$ may have a D or a 0 on that edge. (The D is placed after any inverters present at the inputs.) For each such e_i, there is a path starting at e_i along which the D may propagate to the output. For each D that we are trying to propagate forward through the circuit, there is an associated value $D.s$, which captures timing information associated with the error value. For each D placed on the fanout edges of the primary inputs, $D.s = 0$.

Evaluation of the output of a gate, g, given its inputs, is done in a manner similar to the standard D-calculus [10] with two differences. The first is that the evaluation for each fanout edge, e_i, of g is done separately. Thus, the different fanout edges may evaluate to different values. The second is that the timing information of each error value E (D or $\overline{D}$), *i.e.* $E.s$ is taken into account while evaluating the value for e_i.

The following cases describe the evaluation of the value on e_i, given the values on the inputs to g for some input vector v.

1. There is a controlling value at the input of g: This forces e_i to the corresponding controlled value.

2. Each input of g has a non-controlling value on it: e_i takes on the non-controlled value.

3. Some set of inputs have an error value on them, some (possibly none) have non-controlling values: Note that both D and $\overline{D}$ cannot be present on the inputs of g since we only propagate

either D's or $\overline{D}$'s at a time, but not both. Let E_k be the error values on the inputs and $d(g)$ be the delay of gate g. Let s be as given by the following table:

E	gate	s
D	**or**	$min(E_k.s)$
D	**and**	$max(E_k.s)$
$\overline{D}$	**or**	$max(E_k.s)$
$\overline{D}$	**and**	$min(E_k.s)$

(a) If $s + d(g) + max(e_i) \geq \delta$, then e_i evaluates to the error value E with $E.s = s$.

(b) If $s + d(g) + max(e_i) < \delta$, then e_i evaluates to the error free value, *i.e.*, 1 for a D and 0 for a $\overline{D}$.

It was proven in [5] that the timed D calculus described above would allow the propagation of the multiple stuck-at fault only along paths of length $\geq \delta$. This result, in conjunction with the result that testing for the multiple stuck-at fault is equivalent to checking if any of the paths in Π are responsible for the delay of the circuit as per Proposition 3.1, provides a correct delay computation algorithm.

4 The Overall Algorithm

4.1 Introduction

The theorem and calculus presented in the previous section point us toward a strategy to determine the delay of circuit that uses stuck-at fault test generation algorithms. Note that the timing information restricts the propagation of the fault or line justification along a particular set of paths. In this section we will describe an overall strategy toward timed test generation, used in our timed-test implementation `TrueD-F`, that is reminiscent of the FAN [7] algorithm for stuck-at fault test generation. Rather than generating tests for multiple stuck-at faults at certain input stems in the circuit, we will be generating timed tests for a stuck-at fault at the output of the circuit, while restricting the paths that are sensitized to a 1 (0), satisfying the conditions of Theorem 3.1.

4.2 The Basic Algorithm and Constituent Procedures

The timed test generation algorithm receives as input a single-output combinational logic circuit with associated delays on the links and/or gates in the circuit, a number δ corresponding to the required delay, and a value (1 or 0) corresponding to the logic value at the output for which we are checking the delay. The algorithm answers the question: "Is the delay of the circuit when a logic 1 (0) is asserted at the output greater than or equal to δ ?"

The overall organization of the algorithm in given in Figure 1. The justification list denoted `jlist` contains the core information during any point in the search. The justification list is initialized to the primary output with the appropriate required delay value before the search is begun in `search_1`.

The procedure `search_1` detailed in Figure 2, picks a gate on the justification list, and calls the backtrace procedure to find a primary input whose logical value is currently unknown. The backtrace procedure is described in Section 7. The primary input is set to a 1 value and the implication procedure is called. The implication procedure involves both forward and backward implication which are described in Sections 5 and 6, respectively. The implication procedure may produce a logical or temporal conflict. If no conflict occurs, `search_1` is called recursively. The procedure terminates in success in `search_1` if the justification list is empty. In the case of a logical or temporal conflict in `search_1`, we backtrack to the most recent primary input setting. The primary input is set to the 0 value and `search_2` detailed in Figure 2 is called. Failure results if either the backtrace procedure is unable to find a primary input to set, or if the space has been completely enumerated without success in `search_2`. In the case of conflict or failure, we have to restore the state of the network to what it was immediately prior to the primary input setting that caused this failure. This is done much the same as in test generation algorithms [7], using a *incremental change list* in a restoration routine.

4.3 The Timed Value of a Gate

The `timed_value` for each gate in the combinational logic circuit has three fields, a logic value `timed_value.value`, a lower bound on the delay `timed_value.lower`, and an upper bound on the delay `timed_value.upper`. Initially, all the logic values are 2's (a "2" refers to an unassigned value, it could be either a 0 or a 1), and the lower and upper bounds are the graphical lower and upper bounds, *i.e.* the length of the topologically shortest and longest paths from any input to the gate. As primary inputs are set to known values, these bounds are tightened during the forward simulation due to the sensitization or blocking of paths to these gates. These fields are also changed by backward implication, when we infer required values at particular gates, given the requirement at the primary output.

4.4 The Justification List

During the search gates are added to the justification list during backward implication, and removed from the list during forward and backward implication. The justification list can be viewed as a frontier of gates that corresponds to the remaining gates whose logic/delay values have to be justified by setting more primary inputs. When the justification list becomes empty, the search is successfully completed. The answer to the question "Is the delay of the circuit when a logic 1 (0) is asserted at the output greater than or equal to δ ?" is yes. If the search space has been exhaustively enumerated without the justification list becoming empty, the search has failed. The answer to the question posed is no. The search may be aborted due to excessive backtracking. In this case, the question has remained unanswered.

Note that the same variable `timed_value` is used for both the actual logic/delay values at a gate due to primary input settings, and the required values inferred by backward implication. In the latter case, the gate will be on the justification list. An invariant that is followed throughout the search that defines the meaning of the justification list is as follows: Any gate whose inputs produce the values in the `timed_value` field of the output should not be on the justification list. All gates whose inputs do not produce the values in the `timed_value` field of the output should be on the justification list.

Another point worthy of note is that unlike in logic testing, gates may be added to the justification list more than once during backward implication. This is elaborated on in Section 6.

Having given a general overview of the algorithm, in the subsequent sections the principal steps of timed forward implication, timed backward implication and timed backtrace are treated in detail.

5 Timed Forward Implication

5.1 Timed Implication

The timed implication procedure follows the organization shown in Figure 3. The primary input is set to the given logical value and the effect propagated into the circuit using the `forward_set` routine. The `forward_set` routine schedules the fanout of the changed primary input, and event-driven timed simulation follows in `forward_imply`. This is followed by backward implication. The two procedures are iterated over till the circuit values do not change. In general, setting a gate in the circuit to a particular value during forward or backward implication requires the forward scheduling of all its fanouts, and the backward scheduling of all fanouts that are on the justification list.

Conflicts are detected during both the implication procedures. These conflicts may be time conflicts or logical conflicts. During forward implication, gates may be removed from the justification list. During backward implication, gates may be added to or removed from the justification list.

5.2 Timed Calculus

The timed D-calculus introduced in [5] and reviewed in Section 3 has been slightly modified in this application. Rather than propagate D and $\overline{D}$'s explicitly it was found somewhat more compact to treat all elements of the calculus as Boolean logical values annotated with delays. Those logical values annotated with delays that can, along some path be greater than δ are regarded, but not actually encoded, as D or $\overline{D}$'s. The modified calculus for forward simulation, *i.e.* the calculus used to propagate `timed_value` fields at the inputs of a gate to the output, is presented in Table 1 for a 2-input `and` gate. The three fields correspond to the logic value, the lower bound on the delay value and the upper bound respectively. For instance, if both inputs to the `and` gate are at 0 values, the faster of these values is propagated through. Hence we take the minimum of the lower and upper bounds on the delay values. Similarly for both 1's we take the maximum of the lower and upper bounds. Note that forward simulation of timed values can be used even when both inputs to the `and` gate are 2's. In this case, the lower bound is the minimum of the two lower bounds, and the upper bound is the maximum of the two upper bounds. If all the gates and nets in the circuit are at unknown values, the calculus in forward simulation will result in lower and upper graphical bounds for path lengths to each gate. These bounds are computed and stored in the `timed_value` field prior to beginning the search. This is very useful since a tightening, *i.e.* lowering, of an upper bound on the delay value of a gate close to the inputs of a network can be propagated deep into the network, even if the intermediate gates are all at unknown values. The calculus in the table can be generalized to `or` gates and gates with arbitrary numbers of inputs, similar to the timed D calculus of Section 3.

5.3 Scheduling and Conflict Detection

The forward implication routine is detailed in Figure 4. It is a simple routine that sequentially picks gates from the forward schedule queue and evaluates the gates based on the timed calculus of the previous section.

The routine `forward_set` (also in Figure 4) checks for conflicts once a gate has been evaluated. Logical conflicts correspond to opposing logical values, *i.e.* 1 instead of 0, 0 instead of 1. Note that logical conflicts occur in forward implication because backward implication may have inferred the opposite values at these gates based on the required value at the primary output and primary

v1 → v2 ↓	0	1	2
0	0 MIN(v1.lower, v2.lower) MIN(v1.upper, v2.upper)	0 v2.lower v2.upper	0 MIN(v1.lower, v2.lower) v2.upper
1	0 v1.lower v1.upper	1 MAX(v1.lower, v2.lower) MAX(v1.upper, v2.upper)	2 v1.lower MAX(v1.upper, v2.upper)
2	0 MIN(v1.lower, v2.lower) v1.upper	2 v2.lower MAX(v1.upper, v2.upper)	2 MIN(v1.lower, v2.upper) MAX(v1.upper, v2.upper)

Table 1: Timed Calculus

input settings. Temporal conflicts correspond to the ranges of delay values existing at the gate not intersecting the range of possible values due to the primary input settings. For instance, the lower bound on the required value at the gate (inferred by backward implication) may be larger than the upper bound on the delay produced by forward implication.

If conflicts do not occur, it is possible that a gate on the justification list may have been successfully justified. Successful justification implies both the logical and timed values required at the gate have been produced by the primary input settings.

The delay and logic values at a gate are updated based on the intersection of the values already existing at the gate and the values produced by forward implication. The array `INT` computes the logical value corresponding to the commutative algebra defined by `INT[2][2] = 2`, `INT[2][1] = 1`, `INT[2][0] = 0`, `INT[1][1] = 1` and `INT[0][0] = 0`. Since we have already checked for logical conflicts, these are the only cases that can occur. The delay values at the gate have been evaluated based on the gate inputs using the timed calculus of Table 1. The range of the computed delay values is intersected with the range of the existing delay values at the gate (possibly inferred by backward implication or merely the initial graphical delay bounds). We are guaranteed that the intersection of the ranges is non-empty since we have already checked for time conflicts.

If any changes occur in any of the fields of a gate, the fanouts of the gate are scheduled for forward implication and all fanouts on the justification list are scheduled for backward implication.

6 Timed Backward Implication

6.1 Introduction

Backward implication is tied to the current justification list. It follows the same overall organization as forward implication. The `backward_imply` routine is detailed in Figure 5. The routines `backward_set_1` and `backward_set_0` are called to backward imply a 1 or 0 at the output of the gate under consideration. We will describe these basic routines in the next two sections, assuming that the gate under consideration is an `and` gate. The routines can easily be generalized to arbitrary simple gates.

6.2 Backward Implying a 1

During purely logical backward implication, given an `and` gate with a 1 at its output, one simply assigns logic 1 values to all the fanins (while checking for possible logical conflicts), removes the gate from the justification list if it was on the list, and adds all the fanins (*i.e.* fanin gates) to the justification list if their logical values were previously unknown.

Timed backward implication of a 1 at the output of an `and` gate is more complicated and is described in Figure 6. When the output of an `and` gate is a 1, the slowest of the 1's at its fanins is responsible for the delay at the output of the gate.

As in logical implication, each fanin gate is set to a 1, and if the previous value was unknown, the fanin gate is added to the justification list. If the previous logical value was 0, we have detected a conflict.

Addition to the justification list is performed using the routine `jlist_add_or_modify` described in Figure 7. As mentioned at the end of Section 4, we may be adding a gate to the justification list more than once during the search. The required logical values for the gate at each addition have to be consistent, as do the delay values. Consistency in the delay values simply means that the *intersection* of the two ranges in delay required should be non-empty. For example, we may place a gate on the justification list requiring a logical value of 1 and delay bounds of (2, 6). Later on, we may find that we also require delay bounds of (3, 7). In this case, we will modify the delay bounds of the gate to (3, 6), *i.e.* the maximum of the lower bounds and the minimum of the upper bounds. The above intersection is the same as the computation in the `forward_set` routine (*cf.* Section 5).

We have a required delay value at the output of the gate. The fanins that are guaranteed to be late enough, *i.e.* whose lower delay bound plus the arc delay from the fanin to the gate is greater than the lower bound required at the output of the gate, are found. The fanins that are candidates are those whose upper delay bound plus the arc delay from the fanin to the gate is greater than the lower bound required at the output of the gate. Thus, the fanins which are late are also candidates.

If no candidates exist, we have a time conflict. If at least one fanin exists which is guaranteed to be late enough, we delete the gate from the justification list, since its required time and logical value will be satisfied by placing all its fanins on the justification list. If no such fanin exists we cannot delete the gate from the justification list, because we do not know if any of the fanins is eventually going to satisfy the arrival delay at the gate. Further, we cannot arbitrarily make a decision at this stage regarding the fanin which is eventually going to satisfy the arrival delay. [1]

If the number of candidates is exactly one, we can ask for the required delay on this fanin, place it on the justification list, and delete the gate from the justification list.

6.3 Backward Implying a 0

We have more choices in backward implication of a 0 at the output of an `and` gate, and this restricts the cases where we find unique backward implications. When the output of an `and` gate is a 0, the fanins can be at 0, 1 or 2 values. The fastest of the 0's at its fanins is responsible for the delay at the output of the gate.

As before we walk through the fanins to the gate under consideration. If we find a fanin that is at an unknown value, but is too early to satisfy the delay requirement at the output of the gate, we set the fanin to the non-controlling value of 1. This means that the fanin will not be responsible for the delay at the output of the gate, since the output is at a 0. We place this fanin on the justification list, asking for a logical value of 1, but a -INFINITY delay value, since we do not care as to when this fanin arrives. Inputs that are unknown, but which are candidates for setting to 0 or 1 are detected.

We check for time conflicts in the case of fanins that are already at the 0 logical value but are too early. We detect fanins that are at logic 0 and which are guaranteed to be late enough, and fanins at logic 0 which are not guaranteed to be late enough but are candidates for being responsible for the delay at the output. The candidate fanins are added to the justification list.

If no two or no candidates exist, *i.e.* the inputs are all 1's or early 2's we have a time conflict. If there is exactly one candidate fanin with an unknown value, then we can ask for the required 0 value

[1] If we did make such a decision, we would have to backtrack on this decision if the subsequent search failed. The D-algorithm [10] for test generation employs such a decision making step and in the worst case can backtrack on logical values at every gate in the circuit.

at this fanin, place this fanin on the justification list, and delete the gate from the justification list. If there are no candidates with unknown values, then the candidates with 0 values which have been added to the justification list are guaranteed to collectively satisfy the arrival delay at the output of the gate, and we remove the gate from the justification list.

6.3.1 A Modification to Forward Implication

Given that gates exist on the justification list that have output values which may be inconsistent with their input values, a modification is necessary to the forward implication calculus in order to detect conflicts quickly.

Consider a 3-input `and` gate on the justification list with an output value of 0. Assume that the inputs are at 1, unknown (2) and unknown (2) respectively. When this gate is evaluated using the timed calculus of Table 1, all three inputs are assumed to be responsible for the delay at the output, since the output value according to the calculus is an unknown (2). However, the additional information that we have regarding the output value of the gate, allows us to ignore the 1 input to the gate in forward implication, obtaining a tighter range on the delay values at the output.

In order to ensure that tighter bounds are calculated, while asking for a 0 at the output of a gate during backward implication, we forward schedule the gate as shown in Figure 7.

7 Timed Backtrace

7.1 Introduction

The timed backtrace procedure searches for a path of "unknown" values from a member of the justification list to a primary input that is at an unknown value. The path that is traversed is either a path consisting of gates which all have unknown logical values, or a path consisting of gates with unknown delay or logical values. In order for the timed test generation algorithm to be complete, we have to ensure that the timed backtrace finds any existing unknown path. In logical backtrace, given that the justification list is non-empty, one is guaranteed a successful backtrace to a primary input with an unknown value. However, in our timed backtrace procedure such a path may not satisfy the required delay value for the chosen member of the justification list and we may fail to find an appropriate primary input.

The overall organization of the timed backtrace procedure is given in Figure 9. The backtrace routine calls `backtrace_0` if the output of the current gate is a 0, or calls `backtrace_1` if the output of the current gate is a 1 or an unknown (2). At any given level of recursion in the backtrace, an invariant is that the delay being asked for at the gate under consideration is always less than the upper bound on the possible delay at the gate. The delay asked for may be greater than the lower bound on the possible delay at the gate. In this case, we are backtracing along a logically unknown path.

7.2 Backtracing a 1 or a Unknown (2)

The backtrace procedure for backtracing a 1 is given as routine `backtrace_1` in Figure 10. We have two lists, `list_u` corresponding to the list of unknown fanins, and `list_1` corresponding to the list of fanins with the 1 logical value. Obviously, we cannot have fanins with logical 0, since that corresponds to a logical conflict.

The fanins to the gate are traversed and the two lists are constructed. If the fanin is on the justification list and the required delay at the gate output minus the arc delay from the fanin to the gate output (call it the required delay at the fanin) is within the delay bounds of the fanin, the fanin is placed on `list_1`. Note that we cannot have a fanin with an unknown logical value on the justification list – we always ask for a particular logical value before we place gates on the justification list. If the fanin is not on the justification list, then if its value is unknown it is placed on `list_u`, else if the required delay at the fanin (computed as before) is within the delay bounds of the fanin it is placed on `list_1`. Simultaneously, we are checking the fanins which are not on the justification list to see if the delay at the output of the gate is already met by the fanins. [2] Recall that if the output of the `and` gate is a 1, the slowest 1 at the inputs is responsible for the delay of the circuit. If any of the fanins have a lower bound on their delay value that is greater than their required arrival time, we deem that the delay at the output of the gate has been met.

If the delay at the output of the gate has been met, we try to backtrace unknown logical values. If the delay has not been met, we try to backtrace along paths so as to meet the delay requirements. To elaborate on the above, if the delay has been met, we simply assign logical 1's to the members of `list_u` and recur to the next level of backtrace, with a -INFINITY delay requirement. If the delay has not been met we first recur on the members of `list_1` (candidates for meeting the delay requirement), and then recur on members of `list_u` that are not guaranteed to meet the delay requirement. The delay value for the next level of recursion is computed by subtracting the arc delay from the fanin

[2] We exclude the fanins on the justification list, since their `timed_value` field has been inferred initially from backward implication and the current partial setting of primary inputs may not actually produce these delay values.

to the gate output from the current delay value.

7.3 Backtracing a 0

Backtracing a 0 is similar to the above procedure. The details of the procedure are given in Figure 11. As before the fanins to the gate are traversed and two lists are constructed. The two lists correspond to the fanins with 0 logical values, namely those in `list_0` and fanins with unknown logical values, namely those in `list_u`.

We check to see if all the fanins with 0 logical values not on the justification list meet the required delay value. Recall that the fastest of these 0's is responsible for the delay at the gate output. If *each* of these 0 fanins has a lower bound greater than the required delay at the fanin (computed by subtracting the arc delay from the fanin to the gate output from the required delay at the gate), we deem the delay at the gate output as having been met. We have ignored the 0 fanins on the justification list during this check for the same reason as in backtracing a 1.

If we have met the delay, we recur on members from `list_u` with purely logical requirements, else we recur on members from `list_0` and `list_u` with appropriate required delays.

With this discussion of the backtrace procedure we have completed our detailed treatment of the timed-test algorithm and its implementation in `TrueD-F`. In the following section we present the results of running `TrueD-F` on a number of examples.

8 Results

We present a comprehensive set of results using an implementation of the algorithms described in this paper in the program `TrueD-F`. All circuits in the proposed TAU-92 benchmark set for delay computation were run. These circuits are described in terms of `NOR` gates. The TAU-92 benchmark set contains the ISCAS-85 combinational logic benchmark set. The examples **cbp.12.2** and **cbp.16.4** are carry bypass adders, where the first number corresponds to the number of bits, and the second to the number of blocks. The examples **false16cla** and **false32rpl** are high-level synthesis examples, corresponding to input and output multiplexed carry-lookahead and ripple-carry adders. The example **tau92ex1** is an optimized random logic circuit. The statistics of the benchmarks are given in Table 2. The number of inputs, outputs, and gates are indicated for each example.

In Table 3, the results obtained by running the timed test generation algorithm for true delay computation are presented. All the examples in the benchmark set were completed within reasonable

EX	inputs	outputs	literals	longest
C17	5	2	19	5.00
C432	36	7	405	19.00
C499	41	32	977	25.00
C880	60	26	718	20.00
C1355	41	32	1121	27.00
C1908	33	25	1225	34.00
C2670	233	140	1764	25.00
C3540	50	22	2332	41.00
C5315	178	123	3923	46.00
C6288	32	32	4752	123.00
C7552	207	108	5488	38.00
cbp.12.2	25	13	331	40.00
cbp.16.4	33	17	413	44.00
false16cla	65	32	653	84.00
false32rpl	129	64	1510	135.00
tau92ex1	135	107	1234	27.00

Table 2: Statistics of Benchmark Examples

amounts of CPU time. The CPU times are on a SUN-4 workstation. Some of these examples have literally millions of paths. However, the "sets of paths" paradigm followed in our delay computation algorithm allows for efficient false path identification. The strategies of [9, 4, 6] do not complete within reasonable times for the large circuits (notably the multipliers) because they operate on a per path basis. More recent implementations of the procedures of [6] that use some notions from [5] perform well on all the circuits under a unit delay model. However, their performance may suffer when a large number of paths have distinct delays that are close to the true delay of the circuit. The performance of the timed test generation strategy is not affected to a first order by the number of distinct path lengths $\geq \delta$ or $< \delta$ when answering the question "Is the delay of the circuit greater than or equal to δ?". Using different gate delays in the benchmark examples results in negligible differences in CPU times required for delay computation.

The large number of backtracks for the carry bypass adders and **tau92ex1** can be explained by the large number of checks required (over 100 checks for the adders). The examples in Table 3 that require a large number of backtracks per check are **C1908**, **C2670** and **C6288**. The example **C2670** has an output which is stuck-at fault redundant, and which feeds the output with critical delay. This "global" redundancy results in a large number of backtracks. In fact, delay computation on an irredundant version of the same circuit using `TrueD-F` requires less than 100 total backtracks.

EX	value	longest delay	true delay	#check	CPU seconds	total backtracks
C17	1	5.00	5.00	1	0.01	0
C432	1	19.00	19.00	1	0.06	1
C499	1	25.00	25.00	1	0.37	1
C880	1	20.00	20.00	1	0.59	31
C1355	1	27.00	27.00	1	0.39	0
C1908	1	34.00	31.00	21	3674.52	89436
C2670	0	25.00	24.00	2	200.23	5306
C3540	0	41.00	39.00	10	181.63	3941
C5315	1	46.00	45.00	9	5.15	116
C6288	1	123.00	122.00	2	802.60	10345
C7552	1	38.00	37.00	9	5.89	61
cbp.12.2	1	40.00	23.00	153	233.27	31134
cbp.16.4	1	44.00	27.00	165	238.24	33454
false16cla	1	84.00	47.00	541	39.63	542
false32rpl	1	270.00	135.00	2253	343.92	2287
tau92ex1	1	27.00	24.00	63	217.48	33530

Table 3: Results on Benchmark Examples

The examples **C1908** and **C6288** have several stuck-at fault redundancies, and also have a very large number of paths with equal delays, many of which are false. These false paths do not necessarily share long sub-paths. While sets of these paths are proven to be false simultaneously, many such sets have to be considered, resulting in the large number of backtracks. Again, the number of backtracks are reduced when irredundant versions of the above circuits are tested for their true delays.

9 Summary and Future Work

We presented details on `TrueD-F`, a sophisticated implementation of timed test generation that uses ideas from the stuck-at fault test generators PODEM [8] and FAN [7]. Timed forward implication, timed backward implication and timed backtrace procedures were developed in a search strategy that determines the truth or falsity of sets of paths simultaneously. In addition to using a novel, and correct, floating delay criterion, the implementation described here significantly outperforms other previous delay computation methods.

Ongoing work includes the development of timed learning techniques to further improve the performance of the timed test generation algorithms which compute the floating delay of a circuit, as well as the application of timed test generation in transition delay computation.

10 Acknowledgments

Thanks to Tony Ma and Rick Rudell for interesting discussions on delay computation. This research was supported in part by the Defense Advanced Research Projects Agency under contract N00014-91-J-1698, and in part by a grant from IBM Corporation.

References

[1] J. Benkoski, E. Meersch, L. Claesen, and H. De Man. Efficient Algorithms for Solving the False Path Problem in Timi ng Verification. In *The Proceedings of the International Conference on Computer-Aided Design*, pages 44–47, 1987.

[2] D. Brand and V. Iyengar. Timing analysis using functional analysis. *IEEE Transactions on Computers*, 37(10), October 1988.

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

[4] H. C. Chen and D. H. Du. Path sensitization in critical path problem. In *Proceedings, Tau 90: 1990 ACM Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, August 1990.

[5] S. Devadas, K. Keutzer, and S. Malik. Delay Computation in Combinational Circuits: Theory and Algorithms. In *The Proceedings of the International Conference on Computer-Aided Design*, pages 176–179, November 1991.

[6] P. C. McGeer *et al.* Timing analysis and delay-fault test generation using path recursive functions. In *Proceedings of the Int'l Workshop on Logic Synthesis*, May 1991.

[7] H. Fujiwara. *Logic Testing and Design for Testability.* MIT Press, Cambridge MA, 1985.

[8] P. Goel. An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits. *IEEE Transactions on Computers*, C30(3):215–222, March 1981.

[9] P. C. McGeer and R. K. Brayton. *Integrating Functional and Temporal Domains in Logic Design.* Kluwer Academic Publishers, 1991.

[10] J. P. Roth. Diagnosis of Automata Failures: a calculus and a method. *IBM journal of Research and Development*, 10:278–291, July 1966.

```
timed_test(po, delay, lvalue) {

    v.value = lvalue;
    v.lower = delay;
    v.upper = INFINITY;
    jlist_add_or_modify(po, v, jlist);
    backward schedule po ;

    status = search_1(jlist);
    return status;
}
```

Figure 1: Timed Test Generation Algorithm

```
search_1(jlist) {

    if length of jlist is zero return SUCCEED;

    gate = first gate in jlist;
    value = gate.timed_value.value;
    delay = gate.timed_value.lower;

    if (backtrace(gate, value, delay, &pi, &pi_value) == FALSE)
         return FAILED;

    if (imply(pi, pi_value, jlist) != IMPLY_CONFLICT) {
        search_status = search_1(jlist);
        if (search_status == FAILED) {
            restore the state of the network to what it was
            prior to the most recent primary input assignment;
            search_status = search_2(jlist, pi, 1 - pi_value);
        }
    } else {
        restore the state of the network;
        search_status = search_2(jlist, pi, 1 - pi_value);
    }
    return search_status;
}


search_2(jlist, pi, pi_value) {

    backtracks = backtracks + 1;
    if (backtracks > BACKTRACK_LIMIT) return ABORTED;

    if (imply(pi, pi_value, jlist) != IMPLY_CONFLICT) {
        search_status = search_1(jlist);
        if (search_status == FAILED)
            restore the state of the network;
    } else {
        search_status = FAILED;
        restore the state of the network;
    }
    return search_status;
}
```

Figure 2: The Search Procedures

```
imply(pi, pi_value, jlist) {

    v = pi.timed_value;
    v.value = pi_value;
    status = forward_set(pi, v, jlist);

    while (status == IMPLY_NORMAL) {
        status = forward_imply(jlist);
        if (status != IMPLY_CONFLICT)
            status = backward_imply(jlist);
    }
    return status;
}
```

Figure 3: The Implication Procedure

```
forward_imply(jlist) {

    status = IMPLY_NOCHANGE;
    while (there is a gate in the forward queue) {
        Evaluate the new value v of the gate using the timed calculus;
        s = forward_set(gate, v, jlist);

        if (s == IMPLY_CONFLICT) {
            empty the forward and backward queues;
            return IMPLY_CONFLICT;
        } else if (s == IMPLY_NORMAL) {
            status = IMPLY_NORMAL;
        }
    }
    return status;
}

forward_set(gate, value, jlist) {

    if ( gate.timed_value.value == 1 - value.value ||
         gate.timed_value.upper < value.lower ||
         gate.timed_value.lower > value.upper )
        return IMPLY_CONFLICT;

    if (gate.in_jlist == TRUE) {
        if ( value.value ==  gate.timed_value.value &&
             value.lower >= gate.timed_value.lower)
            delete gate from jlist;
    }

    new_value.value = INT[value.value][gate.timed_value.value];
    new_value.upper = MIN(gate.timed_value.upper, value.upper);
    new_value.lower = MAX(gate.timed_value.lower, value.lower);
    if (gate.timed_value == new_value) return IMPLY_NOCHANGE;

    for each fanout of gate, fo {
        forward schedule fo;
        if (fo.in_jlist == TRUE)
            backward schedule fo;
    }
    return IMPLY_NORMAL;
}
```

Figure 4: Forward Implication

```
backward_imply(jlist) {

    status = IMPLY_NOCHANGE;
    while (there is a gate in the backward queue) {
        if (gate.value == 1) s = backward_set_1(gate, jlist);
        else s = backward_set_0(gate, jlist);

        if (s == IMPLY_CONFLICT) {
            empty the forward and backward queues;
            return IMPLY_CONFLICT;
        } else if (s == IMPLY_NORMAL) {
            status = IMPLY_NORMAL;
        }
    }
    return status;
}
```

Figure 5: Backward Implication

```
backward_set_1(gate, jlist) {

    s = IMPLY_NOCHANGE;
    num_late_fanin = num_cand_fanin = 0;
    for each fanin of gate, fanin {

        value = fanin.timed_value;
        value.value = 1; delay = arc delay from fanin to gate;

        if (fanin.timed_value.value == 1 - value.value)
            return IMPLY_CONFLICT;
        else if (fanin.timed_value.value == 2)
            s = jlist_add_or_modify(gate, fanin, value, jlist);

        if (gate.timed_value.lower <= value.lower + delay)
            num_late_fanin = num_late_fanin + 1;
        if (gate.timed_value.lower <= value.upper + delay) {
            num_cand_fanin = num_cand_fanin + 1;
            cand_fanin = fanin;
        }
    }

    if (num_cand_fanin == 0) return IMPLY_CONFLICT;
    else if (num_late_fanin > 0) {
        s = IMPLY_NORMAL;
        delete gate from jlist and forward schedule gate;
    } else if (num_cand_fanin == 1 && num_late_fanin == 0) {
        delete gate from jlist and forward schedule gate;

        value = cand_fanin.timed_value;
        delay = arc delay from cand_fanin to gate;
        value.lower = gate.timed_value.lower - delay;
        value.upper = INFINITY;

        s = jlist_add_or_modify(gate, fanin, value, jlist);
    }
    return s;
}
```

Figure 6: Backward Implying a 1

```
jlist_add_or_modify(gate, fanin, value, jlist) {

    new_value.value = INT[value.value][fanin.timed_value.value];
    new_value.upper = MIN(fanin.timed_value.upper, value.upper);
    new_value.lower = MAX(fanin.timed_value.lower, value.lower);
    backward schedule fanin;
    if (fanin.value == 0) forward schedule fanin;
    for (each fanout of fanin, fo) {
        if (fo != gate) forward schedule fo;
    }
    fanin.timed_value = new_value;
    add fanin to the jlist if it is not on it;
    return IMPLY_NORMAL;
}
```

Figure 7: Adding Gates To the Justification List

```
backward_set_0(gate, jlist) {

    s = IMPLY_NOCHANGE;
    num_fanin_0 = num_fanin_2 = 0;
    for (each fanin of gate, fanin) {
        value = fanin.timed_value;
        delay = arc delay from fanin to gate;

        if (value.value == 2) {
            if (gate.timed_value.lower > value.upper + delay) {
                value.value = 1;
                value.lower = -INFINITY; value.upper = INFINITY;
                s = jlist_add_or_modify(gate, fanin, value, jlist);
            } else {
                fanin_2 = fanin; num_fanin_2 = num_fanin_2 + 1;
            }
        }
        else if (value.value == 0) {
            if (value.upper + delay < gate.timed_value.lower)
                return IMPLY_CONFLICT;
            else if (value.lower + delay >= gate.timed_value.lower)
                num_fanin_0 =  num_fanin_0 + 1;
            else {
                num_fanin_0 =  num_fanin_0 + 1;
                value.lower = gate.timed_value.lower - delay;
                value.upper = INFINITY;
                s = jlist_add_or_modify(gate, fanin, value, jlist);
            }
        }
    }
    if (num_fanin_0 == 0 && num_fanin_2 == 0) return IMPLY_CONFLICT;
    else if (num_fanin_0 == 0 && num_fanin_2 == 1) {
        delay = arc delay from fanin_2 to gate;
        value.value = 0;
        value.lower = gate.timed_value.lower - delay;
        value.upper = INFINITY;
        delete gate from jlist and forward schedule gate;
        s = jlist_add_or_modify(gate, fanin_2, value, jlist);
    }
    else if (num_fanin_2 == 0) {
        s = IMPLY_NORMAL;
        delete gate from jlist and forward schedule gate;
    }
    return s;
}
```

Figure 8: Backward Implying a 0

```
backtrace(gate, value, delay, pi, pi_value) {

    if (gate is a primary input) {
        *pi = gate;
        *pi_value = value;
        return TRUE;
    } else {
        if (value == 0) return backtrace_0(gate, delay, pi, pi_value);
        else return backtrace_1(gate, delay, pi, pi_value);
    }
}
```

Figure 9: Backtrace

```
backtrace_1(gate, delay, pi, pi_value) {

    delay_met = FALSE;
    for (each fanin of gate, fanin) {
        newd = delay - arc delay of fanin to gate;
        v = fanin.timed_value;

        if (fanin.in_jlist == TRUE) {
            if (newd >= v.lower && newd <= v.upper) {
                append fanin to list_1;
            } else {
                if (newd <= v.lower) delay_met = TRUE;
                if (v.value == 2)
                    append fanin to list_u;
                else if (newd >= v.lower && newd <= v.upper)
                    append fanin to list_1;
            }
        }
    }

    status = FALSE;
    if (delay_met == TRUE) {
        newv = 1; newd = -INFINITY;
        for (each member in list_u, fanin) {
            if (backtrace(fanin, newv, newd, pi, pi_value) == TRUE) {
                status = TRUE;
                break;
            }
        }
    } else {
        for (each member in list_1 and list_u, fanin) {
            newv = 1; newd = delay - arc delay of fanin to gate;
            if (fanin.timed_value.upper >= newd) {
                if (backtrace(fanin, newv, newd, pi, pi_value) == TRUE) {
                    status = TRUE;
                    break;
                }
            }
        }
    }
    return status;
}
```

Figure 10: Backtracing a 1 or a 2

```
backtrace_0(gate, delay, pi, pi_value) {

    seen_0 = FALSE;
    delay_not_met = FALSE;
    for (each fanin of gate, fanin) {
        v = fanin.timed_value;

        if (fanin.in_jlist == TRUE) {
            if (newd >= v.lower && newd <= v.upper)
                append fanin to list_0;
        } else {
            if (v.value == 0) {
                seen_0 = TRUE;
                newd = delay - arc delay of fanin to gate;
                if (v.lower < newd) delay_not_met = 1;
                if (newd >= v.lower && newd <= v.upper)
                    append fanin to list_0;
            } else if (v.value == 2)
                append fanin to list_u;
        }
    }

    status = FALSE;
    if (seen_0 == TRUE && delay_not_met == FALSE) {
        newv = 1; newd = -INFINITY;
        for (each member in list_u, fanin) {
            if (backtrace(fanin, newv, newd, pi, pi_value) == TRUE) {
                status = TRUE;
                break;
            }
        }
    } else {
        for (each member in list_0 in list_u, fanin) {
            newv = 0; newd = delay - arc delay of fanin to gate;
            if (fanin.timed_value.upper >= newd) {
                if (backtrace(fanin, newv, newd, pi, pi_value) == TRUE) {
                    status = TRUE;
                    break;
                }
            }
        }
    }
    return status;
}
```

Figure 11: Backtracing a 0