Performance Validation Through Implicit Removal of Infeasible Paths of the Behavioral Description

Dheepakkumaran Jayaraman, Spyros Tragoudas
ECE Dept., Southern Illinois University, Carbondale,
Carbondale, IL-62901,E-mail: {jayarama,spyros}@engr.siu.edu

Abstract— In this paper we present a novel algorithm to identify infeasible paths in the behavioral code. The proposed approach initially partitions the behavioral code into segments. At each code segment it stores feasible paths implicitly. It also stores collections of input assignments which are derived using selected statements in the code segment. The method requires state-of-the-art data structures to store feasible paths and the required functions. Experimental results demonstrate the scalability of the proposed method.

Keywords—BDD, timing analysis, timing optimization, code optimization

1 Introduction

Validation is increasingly perceived as the major bottleneck in integrated circuit design. For most designs, one of the hardest problems is evaluating the performance of the initial behavioral description coded into a hardware description language (HDL). In HDL, a path is an unique sequence of branches from the function entry to the exit (not to be confused with logical paths in circuits). Paths that do not reach an output will be referred to as sub-paths. In this paper, the term sub-path is also referred as path. A path is feasible (or testable) if there exists input assignment which causes the path to be traversed during program execution. Otherwise, the path is infeasible (untestable).

The challenge in performance validation is to keep track of feasible paths in the behavioral code. The number of these paths is enormous, especially in the presence of embedded loop structures in the behavioral code. For this reason, it is proposed that the code is partitioned into segments and the feasible paths for each segment are kept separately.

The problem of identifying infeasible paths is different from test generation for behavioral HDL models [1], [2]. Paoli et. al [3] proposed a path enumerative method that uses Constraint-Based test to identify infeasible paths in the behavioral code. Verma et. al [4] proposed a heuristic algorithm to identify infeasible paths in the behavioral code across processes. Altenbernd [5] uses a combination of path enumeration, path pruning, and symbolic evaluation to find infeasible paths. Kountouris [6] studies detection of infeasible paths in the synchronous real-time language. Liu et al. [7] use symbolic evaluation of higher languages to avoid infeasible paths. Stenstrom [8] find loop bounds and infeasible paths by symbolic simulation on the binary code. Healy et al. [9] use value-dependent constraints to find infeasible paths.

Yang et al. [10] uses SAT solver to identify infeasible paths and their removal of infeasible paths is implicit. The method in. [11] use binary decision diagrams to identify infeasible paths implicitly. However both are limited to acyclic codes.

The proposed approach efficiently remove infeasible paths implicitly in cyclic code. At each statement appropriate functions are defined and used for infeasible path identification. The idea of using functions to identify infeasible paths was introduced in [11] but the function generation is optimistic and removed paths are indeed feasible. The proposed method guarantees that whenever the algorithm labels a path as infeasible then there is no input assignment to activate the statements along the path. The feasible paths at each hierarchy are kept in a compressed form through appropriate data structures [12], [13] to ensure scalability. In contrast to the hierarchical treatment in architectural level synthesis (hierarchy is enabled at each control flow statement) our approach defines hierarchies only for loop structures.

The paper is organized as follows: Section **2** presents an approach to generate two functions for each program statement to guide infeasible path identification. Section **3** describes infeasible path identification algorithm. Section **4** presents experimental results on large benchmarks to demonstrate the effectiveness of the approach. Section **5** concludes.

2 Function Generation

This section shows an implicit way to generate the functionality of the synthesizable hardware description language. The functions are generated for each statement in the code. Function F^L (resp. F^H) consists of a subset of (resp. superset) input configurations that comprise the functionality of the statement. These functions contains all the possible input configuration that activates the statements through any possible paths or sub-paths.

The HDL is represented using Control Data Flow Graphs (CDFGs) [14], [15]. The CDFG graph is traversed to implicitly generate functions for the data flow and control flow. The nodes (also called basic block) in the CDFG are either operation nodes or condition nodes or loop nodes. An operation node corresponds to operation such as *and*, *or*, *xor*, =. A condition node models conditional structures, such as $if - then - else$ statement. A loop node

This research has been supported in part by the NSF I/UCRC for Embedded Systems at SIUC. This material is based upon work supported by the National Science Foundation under Grant No. 0856039. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

models as *loop* statements.

A. Function for statements

Function generation for primitive operation nodes such as *and*, *or* is straight forward. For example, consider a statement $x := a$ *and* b, the function F_x^L is generated as $F_a^L \wedge F_b^L$ and the function F_x^H is formed as $F_a^H \wedge F_b^H$.

For complex operation nodes such as xor,$xnor$, the statement is expressed as sum-of-products (SOP) and the functions are generated using the SOPs. For example, consider a statement $x := a$ *xor* b, the SOP is $x := a\overline{b} + b\overline{a}$. Using the SOP, functions F_x^L is generated as $(F_a^L \wedge F_b^H) \vee (F_b^L \wedge F_a^H)$. If the SOP contains complementary literal then negation function is used for that literal. In this case, the superset function is used for the complementary literal $\overline{b}$. Similarly function F_x^H is generated as $(F_a^H \wedge F_b^L) \vee (F_b^H \wedge F_a^L)$.

B. Function for Conditional Statements

The following example illustrates how to generate functions for the if-*then-else* structure using the example in Fig. 1. A similar procedure is followed for if-*then-elsif* and *case* structures.

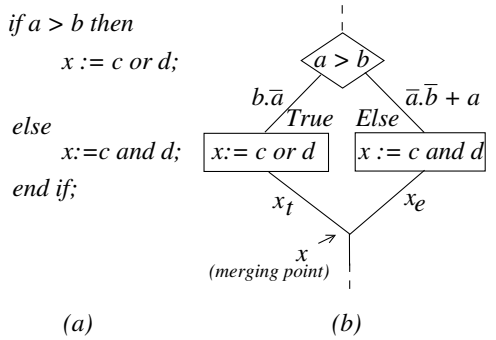

Fig. 1. VHDL code segment of if-then-else structure.

The function generation for if $(a > b)$ *then* structure consists are two *ctrl* edges (true and false). Both true and false *ctrl* edges are expressed as separate SOPs. The SOP for true edge is $b\overline{a}$ and the SOP for the false edge is $\overline{a}\overline{b} + a$. The true edge function has F^L and F^H functions defined as $F^L_{ctrl_t} = (F_b^L \wedge F_a^H)$ and $F^H_{ctrl_t} = (F_b^H \wedge F_a^L)$. Similarly, the false *ctrl* edge functions are $F^L_{ctrl_f} = (F_a^H \wedge F_b^H) \vee F_a^L$ and $F^H_{ctrl_f} = (F_a^L \wedge F_b^L) \vee F_a^H$.

Then the body of the if-*then-else* structure is processed. Let variable x_t is used to explicitly denote the *true* body. The functions for x_t as $F^L_{x_t} = F^L_{ctrl_t} \wedge (F_c^L \vee F_d^L)$ and $F^H_{x_t} = F^H_{ctrl_t} \wedge (F_c^H \vee F_d^H)$.

Then the *else* body statement is processed. Let variable x_f denote the $false$ body. The two functions of this statements are generated $F^L_{x_f} = F^L_{ctrl_f} \wedge (F_c^L \wedge F_d^L)$ and $F^H_{x_f} = F^H_{ctrl_f} \wedge (F_c^H \wedge F_d^H)$.

Finally at the merging point (statement x), function F_x is formulated using the functions $F^L_x = F^L_{x_t} \wedge F^L_{x_f}$ and $F^H_x = F^H_{x_t} \vee F^H_{x_f}$ and it is stored in the output edge x of the condition node. If the function F_x^H is empty, then all the paths through the statement can be removed as infeasible. These statements will never gets executed under any circumstance and it is a dead code.

For every statement i, the approach in [11] allocates only the superset function F_i^H. The functions that are formed for conditional statements cannot always guarantee infeasible paths were removed. In particular a conditional statement that is executed later may inadvertently prevent the generation of all patterns that sensitize either the *true* or *else* branch. This can lead to incorrect identification of infeasible paths through the respective branch.

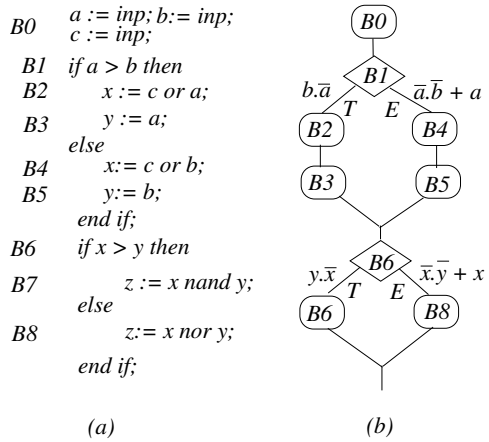

Fig. 2. VHDL code segment of if-then-else structure.

This is explained with the example shown in Fig. 2. Initially, the condition structure is traversed to generate the superset functions for x_t by processing $B1$ (SOP), $B2$ and $B3$. Similarly, x_f is generated by processing $B1$ (SOP), $B4$ and $B5$. At the merging point of if-*then-else* structure, an union operation on true and false edge functions is performed, $F^H_x = F^H_{x_t} \vee F^H_{x_f}$. Similarly F^H_y is also generated. The functions F^H_x and F^H_y contains maximum number of input configurations through the edges x and y.

When subsequent if-*then-else* structure is processed, function generation involving complemented F^H_x or F^H_y may result in empty function. In the example when $x > y$ is processed, functions are generated for the true edge SOP $y\overline{x}$ as $F^H_{ctrl_t} = (F_y^H \wedge \neg F_x^H)$. Here, $F^H_{ctrl_t}$ is an empty function. The approach in [11] will identify all the paths through this edge as infeasible.

Using the proposed approach, $F^H_{ctrl_t}$ is generated with a subset function on complementary literals. $F^H_{ctrl_t}$ is not an empty function and all the paths through this edge are not reported as infeasible.

C. Functions for Loop Structures

The remaining illustrates the function generation procedure for *loop* structures. Each loop structure is processed by repeatedly forming functions for the loop *condition* and the loop *body* (see also Fig. 3). At each iteration, two functions are generated for each edge i. Let $F^L_{i_k}$ (resp. $F^H_{i_k}$) denotes the subset of (resp. superset) on number of input

configurations that comprise the functionality of the statement (edge) i on the k^{th} iteration. Let $F^L_{i_a}$ (resp. $F^H_{i_a}$) be the function for an edge i generated from the code preceding the while loop structure. Let the loop condition (entry) edge in the k^{th} iteration be denoted as $F^L_{lctrl_k}$ and $F^H_{lctrl_k}$.

We illustrate the formulation of functions using Fig. 3. First, *lctrl*, is processed and the *lctrl* edge (entry) SOP is expressed as $\overline{xy} + x$. Using the SOPs the functions are generated as $F^L_{lctrl_1} = (F^H_{x_a} \wedge F^H_{y_a}) \vee F^L_{x_a}$ and $F^H_{lctrl_1} = (F^L_{x_a} \wedge F^H_{y_a}) \vee F^H_{x_a}$.

Secondly, the loop *body* is processed and function of the edge x is generated. Here, the edge x is a condition node and the function generation is similar to the previous example. Let $F^L_{x_{c1}}$ and $F^H_{x_{c1}}$ be the functions generated for the x edge of the condition node.

Finally, function F_x for the first iteration is formulated as $F^L_{x_1} = F^L_{lctrl_1} \wedge F^L_{x_{c1}}$ and $F^H_{x_1} = F^H_{lctrl_1} \wedge F^H_{x_{c1}}$ and it is stored in the output edge x of the loop node.

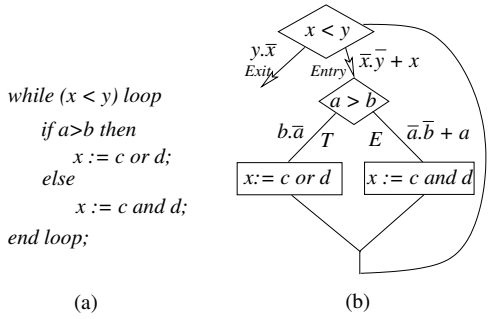

Fig. 3. Code segment of while loop structure.

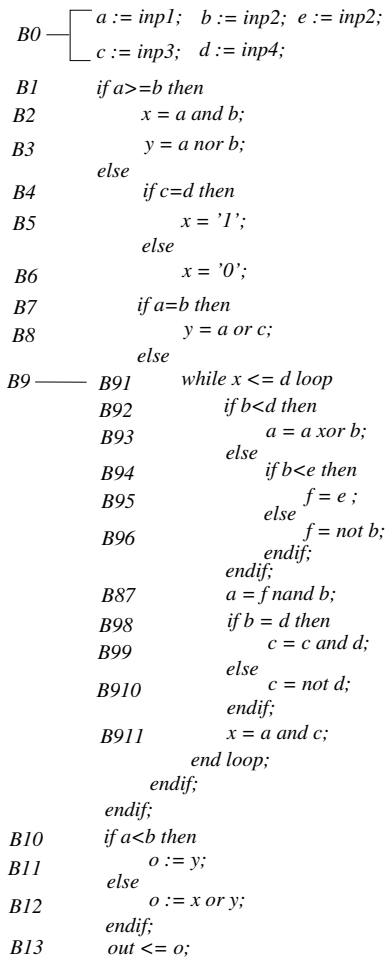

Fig. 4. Code for while loop structure and its Basic Blocks.

For the second and consecutive iterations, the above procedure is repeated to generate new functions for the edges x. The function generated in the previous iteration for each edge is used as the input for the current iteration. The function generation process is terminated, when all functions generated in the current iteration are identical to the previous iteration.

Then the function F^L_i that serves as a subset will be the intersection of all $F^L_{i_k}$ and the function F^H_i will be the union of all $F^H_{i_k}$, which is $F^L_i = \bigcap_k F^L_{i_k}$ and $F^H_i = \bigcup_k F^H_{i_k}$. Functions F^H_i and F^H_i of each statement i is used to identify infeasible paths through i.

3 Implicit removal of infeasible paths

This section illustrates the proposed implicit method that identifies infeasible paths i.e., paths which are executable according to the control flow graph structure, but not feasible when considering the semantics of the program and the possible inputs. The algorithm operates on the graph by partitioning the code in three segments for each loop: a) basic blocks proceeding the loops, b) basic blocks inside the loops and c) basic blocks after the loop. In each segment the algorithm finds sequences of basic blocks along a path which are never executed together due to conflict in input assignments.

Inputs to this section is the CDFG graph and functions generated for each statement in Section **2**. Execution paths are kept implicitly in a Zero Suppressed Binary Decision diagram (ZBDD). It is demonstrated in [11] that ZBDD are capable of storing implicitly very large number of HDL execution paths. The function F_H for each basic block is used to identify infeasible paths. The algorithm maintains a ZBDD for each basic block that contains paths from some inputs and up to the current basic block.

A path or sub-path is feasible (executable) if and only if the functions associated with each basic block on the path are satisfied simultaneously. In other words, there exists input assignment that satisfies the each basic block on the path simultaneously.

The following five steps determines if a sub-path is infeasible.

Step 1) For each basic block B_i, identify the set of predecessor basic blocks $Spre_{B_i}$, along the same path by a backward depth-first traversal. Initial value for the set $Spre_{B_i}$ is set to {}.

Step 2) All predecessor basic blocks in $Spre_{B_i}$ whose functions F_H have disjoint set of variables with the F_H functions of the current basic block B_i are ignored.

Step 3) All predecessor basic blocks in $Spre_{B_i}$ whose

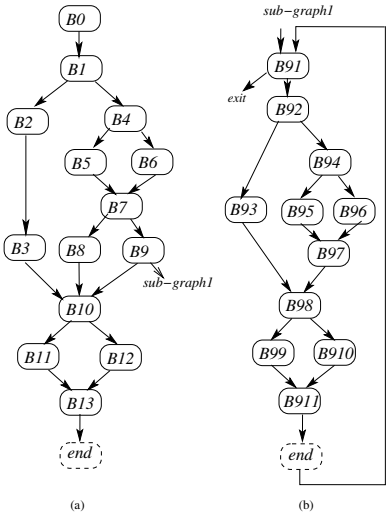

Fig. 5. CDFG of the code in Fig. 4. (a) *O-CDFG* (b) *L-CDFG*

functions F_H cannot be satisfied simultaneously with the function F_H of the current basic block B_i are termed as conflicting to B_i.

Step 4) All paths through conflicting basic blocks are identified and are implicitly removed from the ZBDD of current basic block B_i.

Step 5) Repeat Steps 2 through 4 by enhancing $Spre_{B_i}$ by another predecessor block following a backward depth first search traversal until no more paths are removed.

When $|Spre_{B_i}|$ is one, then pairs of basic blocks are simultaneously checked. When $Spre_{B_i}$ is has a value k > 1, then k + 1 basic blocks must be simultaneously checked in step 3. However, this does not impact the time complexity of the algorithm since many infeasible paths can be efficiently represented by sharing common sub-paths.

In order to ensure scalability, the maximum value of variable $|Spre_{B_i}|$ in the algorithm is always bounded by a small constant. However, our experimental evaluation show that Steps 2 and 3 typically execute very few times, and there is no need for a preset bound on the value of B_{pre}. This ensures that the proposed approach is scalable.

An illustration of Procedure 1 is given by considering the HDL code shown in Fig. 4 and its corresponding CDFG shown in Fig. 5. For a given code, there exists one CDFG graph for codes outside loop and sub-graphs for each loops structures. In the example, the sub-graph that represents the loop statement is accessed through the basic block $B9$ in the main graph.

In the rest of the section, CDFG graphs for loops nodes are denoted as *L-CDFG* and the CDFG graph for codes outside loop are denoted as *O-CDFG*. The code is partitioned and processed in topological order.

In this example, there are three segments. All the basic blocks preceding $B9$ in Fig. 4 form the first segment. All the basic blocks in $B91$ to $B98$ Fig. 4 form the second segment. The of the basic blocks below $B9$ in Fig. 4 form to the third segment.

When the procedure operates on first segment, the basic block $B8$ is processed. The ZBDD Z_{B8} is formed that eventually contain all executable sub-paths up to basic block $B8$. From a given set of basic blocks, Step 2 returns those basic blocks whose function depends on at least one same variable as the function of basic block $B8$ does. The predecessor basic blocks $Spre_{B_i}$ are identified by a depth first search, in this case $Spre_{B_i}$ is $\{B4\}$. Initially, the union of ZBDDs for all the predecessor basic blocks of $B8$ is obtained in a ZBDD $Z_{B_{pre}}$. Then, the current basic block is added into each path in $Z_{B_{pre}}$ to form Z_{B8}, and each path will be implicitly determined as feasible or infeasible.

Let F^H_{B4} and F^H_{B8} be the superset functions of $B4$ and $B8$. The functions of $B4$ and $B8$ do not have disjoint set of variables. The superset functions are checked if they have conflict in input assignment using $F^H_{B4} \wedge F^H_{B8}$. It is observed that $B4$ and $B8$ are conflicting. Then all paths which are passing through $B4$ and $B8$ are implicitly removed from Z_{B8} and will not be further considered. In this example the paths are {B0, B1, B4, B6, B7, B8} and {B0, B1, B4, B5, B7, B8} which is shown in Table. I.

When segment two is processed, the procedure operates on the basic blocks from $B91$ to $B911$ shown in Fig. 4. Now, it is necessary to identify the conflict basic blocks preceding the loop and inside the loop structure. So the algorithm performs the union operation on the superset functions of feasible paths that leads to $B9$. In this example the feasible paths that leads to $B9$ are {B0, B1, B4, B6, B7, B9} and {B0, B1, B4, B5, B7, B9}. Lets denote the function formed after the union operation as $F^H_{B_{pre}}$. The function $F^H_{B_{pre}}$ is used inside the loops to identify conflicts between the basic blocks preceding the code and inside the code.

For each basic block in the loop structure, Procedure 1 is repeated on the superset function. In the example shown in Fig. 4, when $B96$ is traversed, using superset function, it is observed that the basic block $B93$ and $B910$ are conflicting. Then all paths which are passing through $B93$ and $B910$ are implicitly removed and will not be further considered. In this example the paths are {B91, B92, B93, B98, B910}. The identification of infeasible paths in the basic block graph in Fig. 4 is shown in Table. I.

Similarly, when segment three is processed, the procedure operates on the basic blocks below $B9$ shown in Fig. 4. Since this segment is below the loop structure, the procedure preforms an union operation of all the superset functions of the feasible paths inside the loops. Let us denote the function formed after the union operation as $F^H_{B_{loop}}$. The superset function $F^H_{B_{loop}}$ is used below loops to identify conflicts between the basic blocks through $B9$. When the basic block $B10$ is processed, is it observed that $B10$ and $B3$ are conflicting. So all the paths between the basic blocks $B10$ and $B3$ are removed from the ZBDD, which is shown in Table. I.

To analyze the time complexity of the algorithm, let n denote the number of basic blocks and let b denote the maximum number of examined sets of basic blocks. The value of B_{pre} is bounded by a constant value. The number of BDD built-in operators invoked to identify the conflict-

TABLE I

Identification of Infeasible Paths in Fig. 4

Current Basic Block (B_i)	Sub-paths Ending at B_i	Conflicting Basic Block	Feasible Sub-paths Ending B_i
B0	{B0}	None	{B0}
B1	{B0, B1}	None	{B0, B1}
B2	{B0, B1, B2}	None	{B0, B1, B2}
B3	{B0, B1, B2, B3}	None	{B0, B1, B2, B3}
B4	{B0, B1, B4}	None	{B0, B1, B4}
B5	{B0, B1, B4, B5}	None	{B0, B1, B4, B5}
B6	{B0, B1, B4, B6}	None	{B0, B1, B4, B6}
B7	{B0, B1, B4, B6, B7}	None	{B0, B1, B4, B6, B7}
	{B0, B1, B4, B5, B7}	None	{B0, B1, B4, B5, B7}
B8	{B0, B1, B4, B6, B7, B8}	B6 & B8	∅
	{B0, B1, B4, B5, B7, B8}	B4 & B8	∅
B91	{B91}	None	{B91}
B92	{B91, B92}	None	{B91, B92}
B93	{B91, B92, B93}	None	{B91, B92, B93}
B94	{B91, B92, B94}	None	{B91, B92, B94}
	.		
	.		
	.		
B910	{B91, B92, B93, B98, B910}	B93	∅
B10	{B0, B1, B2, B3, B10}	None	{B0, B1, B2, B3, B10}
	{B0, B1, B4, B6, B7, B9, B10}	None	{B0, B1, B4, B6, B7, B9, B10}
	{B0, B1, B4, B5, B7, B9, B10}	None	{B0, B1, B4, B5, B7, B9, B10}
	{B0, B1, B4, B6, B7, B8, B10}	None	{B0, B1, B4, B6, B7, B8, B10}
	{B0, B1, B4, B6, B7, B8, B10}	None	{B0, B1, B4, B6, B7, B8, B10}
B11	{B0, B1, B2, B3, B10, B11}	None	{B0, B1, B2, B3, B10, B11}
	{B0, B1, B4, B6, B7, B9, B10, B11}	B4 & B11	∅
	{B0, B1, B4, B5, B7, B9, B10, B11}	B4 & B11	∅
	{B0, B1, B4, B6, B7, B8, B10, B11}	B4 & B11	∅
	{B0, B1, B4, B6, B7, B8, B10, B11}	B4 & B11	∅

ing sets of basic blocks is $O(b)$. In order to check and store all executable paths in a ZBDD, $O(n \times b)$ calls to BDD and ZBDD operators are required.

4 Experimental Results

We used the following benchmarks to validate the proposed approach: ITC'99 [16], Greatest Common Divisor(GCD), Differential Equation (DIFFEQ). The behavioral descriptions of GCD and DIFFEQ were taken from a benchmark suit of [17] (not to be confused with RTL). These program codes are efficiently translated to CDFGs using the Van VHDL analyzer [18] and the CDFG package [19]. These benchmark are modified so that it can be handled by Van [18] without modifying the functionality. We implemented the proposed methodology in C++ language on a 1.2GHz UltraSPARC workstation with 8-GB RAM. Functions are implemented using the CUDD package [20].

TABLE II

Requirements of the employed Data structures

Bench	# lines	# loops	BDD Nodes	CPU (sec)	ZBDD Nodes	CPU (sec)
b14	840	2	15,625	9,312	524	136
b15	925	4	19,245	1,045	504	122
GCD	772	2	19,548	9,409	325	80
DIFFEQ	845	3	21,348	9,928	412	96

Table II shows the properties of the data structure for different benchmarks. It also shows the memory usage and the time taken by the decision diagram for the benchmarks. Column 1 lists a collection of benchmarks. Column 2 shows the number of VHDL lines. Column 3 shows the number of loops in the benchmarks. Column 4 shows the number of nodes in the binary decision diagram to generate the functionality of the design. For these experiments, we limited the memory allocation by binary decision diagrams to 500Mb. Column 5 shows the CPU time used to generate the function in seconds. Column 6 gives the number of nodes in the Zero-Suppressed binary decision diagram to represent the paths in the benchmark. Column 7 shows the CPU time used to generate the paths in seconds.

TABLE III

Infeasible Paths Identified During Function Generation

Bench	Structural paths	Infeasible paths			
		[11]	(%)	Proposed	(%)
b14	1,643	85	5	26	1.5
b15	1,865	72	3	12	0.6
GCD	2,019	112	6	31	1.5
DIFFEQ	2,603	291	11	81	3

Table III shows the number of infeasible paths identified during function generation in Section **2**. Column 1 lists a collection of benchmarks. Column 2 represents the number of structural paths in the code. Column 3 shows the number of infeasible paths identified by [11]. Column 4 shows the percentage of paths identified. Column 5 shows the number of infeasible paths identified by the proposed method. Column 6 shows the percentage of paths identified by the proposed method.

During function generation procedure, if an superset function of a basic block is empty, then all the paths through the basic block can removed as infeasible. These codes will never get triggered during the execution of the program. From Column 3, it is observed that, [11] removes many feasible paths as infeasible. Column 5 shows the number paths removed by the proposed method during

TABLE IV
INFEASIBLE PATHS DUE TO CONFLICTING INPUT ASSIGNMENTS

Bench	Structural paths	[11]			Proposed				
		Infeasible	%	CPU (sec)	Infeasible	%	CPU (sec)	Feasible	%
b14	1,643	402	24	127	354	21	253	44	11
b15	1,865	298	16	256	171	9	395	50	17
GCD	2,019	817	40	991	621	30	1,822	171	21
DIFFEQ	2,603	928	36	1,405	553	21	2,654	222	24

function formation. All the paths are provable infeasible.

Table IV shows the number infeasible paths identified due to the conflicts in input assignments. Column 1 lists a collection of benchmarks. Column 2 represents the number of structural paths in the code. Column 3 shows the number of infeasible paths identified by [11]. Column 4 shows the percentage of paths identified. Column 5 shows the CPU time.

Column 6 shows the number of infeasible paths identified by the proposed method in Table IV. Column 7 shows the percentage of paths identified. Column 8 shows the CPU time. All the paths listed in column 6 are infeasible, (i.e.) none of the paths are reported as infeasible due to the limitation of the back-end decision process. From column 6-7 it is evident that the proposed method is very effective in removing the infeasible paths. For path intensive benchmark such as GCD, the proposed method has removed 30% paths as infeasible, which is a significant portion.

Column 9-10 shows the number of feasible paths identified as infeasible by [11]. Column 9 shows the number of feasible paths identified by the proposed approach. Column 10 shows the percentage of paths identified by the proposed approach. These paths are identified as infeasible due to optimistic function generation by [11].

5 CONCLUSION

A novel and efficient framework to classify the paths in code as either feasible or infeasible has been presented. Both the data dependency and the control dependency are then considered in order to find the infeasible paths. Experimental results on benchmarks show that the proposed method determined an average that a quarter of the paths are infeasible. The proposed method is illustrated using hardware description language (HDL), however the approach is not limited to HDL.

It is currently investigated whether accurate time bounds can be obtained when Worst Case Execution Time (WCET) is calculated on the feasible paths rather than the original HDL code. The analysis is based on an Integer Linear Programming (ILP) formulation originally proposed by Malik et al. [21].

REFERENCES

[1] S. K. S. Hari, V. V. R. Konda, V. Kamakoti, V. M. Vedula, and K. S. Maneperambil, "Automatic constraint based test generation for behavioral hdl models," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 16, pp. 408–421, April 2008.

[2] F. Ferrandi, F. Fummi, and D. Sciuto, "Implicit test generation for behavioral vhdl models," *In Proc. International Test Conference*, pp. 587 –596, Oct. 1998.

[3] P. Christophe, N. Marie-Laure, S. Jean-Franois, and C. Antoine, "Path-oriented test data generation of behavioral vhdl description," *Electronic Design, Test and Applications, IEEE International Workshop on*, vol. 0, p. 382, 2002.

[4] S. Verma, K. Ramineni, and I. G. Harris, "A control-oriented coverage metric and its evaluation for hardware designs," *Journal of Computer Science*, pp. 302–310, 2009.

[5] P. Altenbernd, "On the false path problem in hard real-time programs," *In Proc. 8th Euromicro Workshop on Real-time Systems*, pp. 102–107, 1996.

[6] A. Kountouris, "Safe and efficient elimination of infeasible execution paths in wcet estimation," *In Proc. 3rd International Workshop on Real-Time Computing Systems and Applications*, pp. 187 –194, oct 1996.

[7] Y. Liu and G. Gomez, "Automatic accurate time-bound analysis for high-level languages," *In Proc. ACM SIGPLAN 1998 Workshop on Languages, Compilers and Tools for Embedded Systems*, pp. 31–40, 1998.

[8] T. Lundqvist, "A wcet analysis method for pipelined microprocessors with cache memories," Ph.D. dissertation, Chalmers University of Technology, Goteborg, Sweden, 2002.

[9] C. Healy and D. Whalley, "Tighter timing predictions by automatic detection and exploitation of value-dependent constraints," *In Proc. 5th IEEE Real-Time Technology and Applications Symposium*, pp. 79–85, 1999.

[10] Z. Yang, B. Al-Rawi, K. Sakallah, X. Huang, S. Smolka, and R. Grosu, "Dynamic path reduction for software model checking," *In Proc. 7th International Conference on Integrated Formal Methods*, 2009.

[11] C. Song and S. Tragoudas, "Identification of critical executable paths at the architectural level," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 27, no. 12, pp. 2291–2302, 2008.

[12] R. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Trans. on Computers*, vol. C-35, no. 8, pp. 677 –691, aug. 1986.

[13] S. Minato, "Zero-suppressed bdds for set manipulation in combinatorial problems," pp. 272–277, 1993.

[14] F. E. Allen, "Control flow analysis," *Proceedings of a symposium on Compiler optimization*, pp. 1–19, 1970.

[15] G. D. Micheli, *Synthesis and Optimization of Digital Circuits*. McGraw Hill, Inc, 1994.

[16] "Itc'99 bechmarks," http://www.cad.polito.it/tools/itc99.html.

[17] "Hlsynth92 benchmark directory at," www.cbl.ncsu.edu/benchmarks/hlsynth92.

[18] S. Park and K. Choi, "Vhdl developers toolkit user guide & reference," Design Automation Lab, Seoul National Univ, Tech. Rep. SNU-EE-TR-1997-5, 1997.

[19] J. Jinhwan, A. Yongjin, and C. Kiyoung, "Cdfg toolkit user guide," Seoul National University, Tech. Rep. SNU-EE-TR-2002-8, 2002.

[20] F. Somenzi, "Cudd: Cu decision diagram package," Department of Electrical, Computer, and Energy Engineering, University of Colorado at Boulder, Tech. Rep., 2008.

[21] Y.-T. S. Li and S. Malik, "Performance analysis of embedded software using implicit path enumeration," *In Proc. 32nd annual ACM/IEEE Design Automation Conference*, pp. 456–461, 1995.