On Timing-Independent False Path Identification

Feng Yuan† and Qiang Xu†‡
†CUhk REliable Computing Laboratory (CURE)
Department of Computer Science & Engineering
The Chinese University of Hong Kong, Shatin, N.T., Hong Kong
‡Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences
Email: {fyuan,qxu}@cse.cuhk.edu.hk

ABSTRACT

This paper is concerned with finding timing-independent false paths that cannot be sensitized under any signal arrival time condition in integrated circuits. Existing techniques regard a path as a true path as long as a vector pair can be found to sensitize it. This is rather pessimistic since such a path might be activated only with illegal states in the circuit and hence it is actually functionally-unsensitizable. In this paper, we develop novel techniques to take the above issue into consideration when identifying false paths, which facilitates us to find much more false paths than conventional techniques. Experimental results on benchmark circuits demonstrate the effectiveness of the proposed methodology.

1. INTRODUCTION

Logic circuits typically contain a large amount of paths down which signals cannot propagate in functional mode, known as *unsensitizable paths*, or simply *false paths* [3, 4]. These paths should not be considered during the design and test of integrated circuits (ICs). For example, static timing analysis (STA) is an integral part in the physical design optimization process (e.g., timing-driven placement) for today's IC designs, used extensively to achieve circuit timing closure. Optimizing false paths, however, does not help to improve the performance of the circuit and the associated cost of optimization and iteration is expensive [1]. Similarly, targeting false paths during manufacturing test is unnecessary and may lead to over-testing of the circuit [9]. Therefore, how to effectively and efficiently identify false paths is an important and relevant problem for IC designers.

False paths can be categorized into three types: (i). *timing-don't-care false paths* with asynchronous or varying time budgets, such as those paths in asynchronous clock domain crossovers; (ii). *timing-independent false paths* that are *logically unsensitizable* in functional mode; and (iii). *delay-dependent false paths*, which are logically sensitizable, but cannot be activated since one or more *on-path signals* are dominated by *side-input signals* all the time. Here the on-path signals refers to those signals that lie on the path being considered, while the side-input signals are other signals driving the logic cells on this path.

Generally speaking, identifying timing-don't-care false paths requires the knowledge of the design and they are typically picked up by designers manually. Various automated false path identification (FPI) techniques have been presented to identify the other two types of false paths (e.g., [4, 6, 9, 13]). When identifying delay-dependent false paths, we need to calculate the signal arrival times to determine whether the on-path signals are always dominated by side-input signals. Consequently, if the circuit timing model is not accurate enough, it is possible that certain true critical paths are claimed to be false and hence are excluded from optimization, leading to a more serious problem of false indication of timing closure and possible silicon failures [6]. This problem is exacerbated with the ever-increasing process variations in advanced semiconductor technology [2], as signal arrival times become statistical values. Therefore, in this work, we focus on identifying timing-independent false paths, which cannot be sensitized under *any* arrival time condition. These paths are guaranteed to be safely removable in circuit timing analysis and manufacturing test.

The path sensitization criteria used in most prior FPI techniques are based on automatic test pattern generation (ATPG) like techniques [13]. To be specific, for a given path, it is assumed to be sensitizable if there exists a test vector pair (v_1, v_2) that activates a transition at the launch point of a path and propagates to its ending point; if, however, we cannot find such a test vector pair in any circumstances, this path is deemed as a false path. Generally speaking, ATPG-based FPI techniques need to exhaustively search in an input space to prove that a targeted path to be false. To alleviate this problem, there have also been some implication-based false path identification techniques (e.g., [5]). Essentially, these methods use the same criteria to identify false paths. The difference is that they try to prove the non-existence of test vectors using implication analysis instead of exhaustive search.

For a particular path, even if we are able to find a test vector pair that activates it, such a test may be *functionally-unreachable*. Consider a finite state machine encoded with one-hot code, the legal combinations of values in the circuit's state elements are only those with a single logic '1' and all the others logic '0'. Consequently, if a path can be activated in this circuit only with multiple logic '1's in these state elements (i.e., containing illegal states), this path is considered to be a true path based on the above criteria, but in fact it is functionally unsensitizable.

Motivated by the above, we propose novel techniques to identify those timing-independent paths that imply illegal states or other logic conflicts when they are activated. To be specific, we adapt the illegal state identification technique presented in [12] and integrate it into our FPI flow. For a given critical path, we present effective and efficient techniques to check whether it is a true path or a false path. We also present novel solutions to address the more general problem of finding as many *false paths* in a circuit as possible. In our experimental results on ISCAS'89 and IWLS'05 benchmark circuits, by injecting the false paths identified with our technique into a commercial static timing verifier, the critical path delay for certain circuits can be significantly reduced, indicating existing STA tools are often over-pessimistic.

The remainder of this paper is organized as follows. Section 2 presents the preliminaries of this work and motivates this paper. In Section 3 and Section 4, we detail the proposed FPI techniques. Experimental results on benchmark circuits are then presented in Section 5 to demonstrate the effectiveness of our technique. Finally, Section 6 concludes this paper.

2. PRELIMINARIES AND MOTIVATION

2.1 Illegal State Identification

Illegal state identification has attracted lots of research interests recently, and a number of techniques have been presented in the literature, including SAT-based methods [8], implication-based strategies [14], mining-based techniques [11], and a recent justification-based method [12].

As [12] is able to effectively and efficiently identify much more illegal states when compared to other techniques, it is utilized in this work. In [12], the authors studied the structural root cause for illegal states and showed that they are mainly caused by multi-fanout nets in the circuit. That is, illegal states would imply logic violations at different branches of the same multi-fanout net, explicitly in the same time frame or implicitly across multiple time frames. Based on this observation, this work defined the so-called justification scheme at every circuit node in the format of $Cube0 \rightarrow 0$ and $Cube1 \rightarrow 1$, denoting that a state cube $Cube0/Cube1$ justifies logic 0/1 on this node. Therefore, illegal states are identified by detecting combinations of justification schemes which can imply contradictory logic values on the same muli-fanout net explicitly or implicitly.

2.2 Motivation

As discussed earlier, existing FPI techniques regard a path to be a true path as long as a test vector pair can be found to sensitize it. However, none of them explicitly considers whether this found vector pair is functionally reachable or not. This is rather pessimistic because: *if a path is activated only with illegal states in the circuit, this path is a false path.*

Let us use the circuit shown in Fig. 1 as an example. Consider the path $P = \{FF1, A, D, F, G, FF4\}$, where a rising transition occurs at the launch point $FF1$ and propagates to the ending point $FF4$. To activate this transition, we need to have an input vector to drive logical values for the on-path signals $\{FF1, A, D, F, G\}$ to be $\{0, 0, 1, 1, 1\}$ in the first

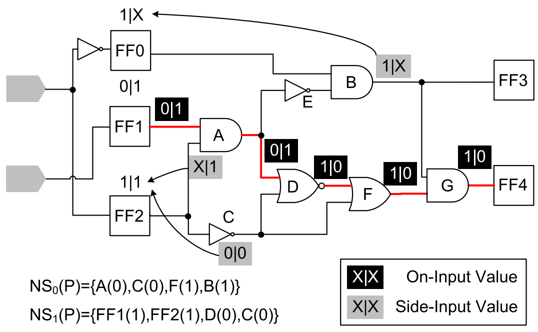

Figure 1: False Path Caused by Illegal State - An Example

clock cycle and $\{1,1,0,0,0\}$ in the second cycle, as shown in the figure. One vector pair on $\{FF0, FF1, FF2\}$, $< 1,0,1; X,1,1 >$, can be found to sensitize this path. Consequently, traditional FPI technique (either ATPG-based or implication-based method) would treat this path as a true path. A closer examination of the vector pair, however, tells us that it is functionally-unreachable. This is because, $FF0$ has to be the inverted value of $FF2$ in functional mode according to the circuit structure, and hence $\{FF0(1), FF2(1)\}$ is an illegal state cube. At this moment, however, we still cannot claim path P is a false path as we are not certain whether there exist other vectors being able to sensitize it without violating functional constraints. Nevertheless, we can find out that $\{FF0(1), FF2(1)\}$ is implied by the transitions occurring on path P through circuit structural analysis, as shown in Fig. 1 (denoted by the arrowed line). In other words, $\{FF0(1), FF2(1)\}$ is a necessary condition to activate path P, and hence we can conclude it is a false path.

The above example motivates us to take illegal states into consideration in false path identification and use implication-based techniques to efficiently determine whether a path is a false path or not.

3. FALSE PATH EXAMINATION CONSIDERING ILLEGAL STATES

In this section, we consider the problem of evaluating whether a given path is a timing-independent false path. This procedure can be integrated into the inner loop of existing circuit optimization tools, e.g., before optimizing the critical path reported by STA tools, we first quickly evaluate whether it is a false path to avoid unnecessary optimization efforts.

3.1 Path Sensitization Criterion

For identifying timing-independent false paths, we have the following theorem:

THEOREM 1. *A path is a timing-independent false path if and only if there exists at least one on-path signal such that when it is a non-controlling value, one or more of its corresponding side-input signals are with controlling values in functional mode.*

PROOF. The sufficiency of this theorem is obvious. That is, when an on-path signal is a non-controlling value while some side-input signals are with controlling values, this path cannot be activated and hence is a false path. As for the necessity of the theorem: (i). when the on-path signal is a controlling value, since we are considering timing-independent false paths where the side-input signals can arrive at any time, the path is sensitizable even if some side-input signals are with controlling values since they can arrive later; (ii). when the on-path signal and its corresponding side-input signals are all non-controlling values, similarly, the side-input signals are likely to arrive earlier so that the path can be activated to be a true path. Therefore, only if when the on-path signal is a non-controlling value while one or more of its side-input signals are with controlling values in functional mode, we can deem this path as a timing-independent false path. ∎

Apparently, with the above theorem, we can derive the following lemma:

LEMMA 2. *A path is not a timing-independent false path if and only if, for any on-path signal, when it is a non-controlling value, all its corresponding side-input signals are also with non-controlling values in functional mode.*

To sensitize a path with either a rising transition or a falling transition at its launch point, all the on-path signals need to have transitions and hence they are applied with both logic '0' and '1' in two consecutive clock cycles. According to the above lemma, for a given path P, to determine whether it is a timing-independent false path, we only need to propagate logic '0' and logic '1' at its launch point separately

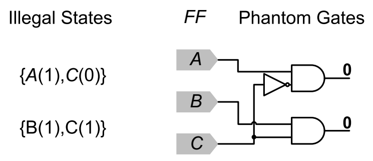

Figure 2: Representation of Illegal States as Phantom Gates

and examine whether those on-path signals with non-controlling values (e.g., logic '1' for AND gate) and their corresponding side-input signals also with non-controlling values can co-exist during propagation in functional mode. They are called *the necessary set-up values to propagate logic '0'* and *the necessary set-up values to propagate logic '1'* for the path, denoted as $NS_0(P)$ and $NS_1(P)$, respectively. To sensitize the path, both $NS_0(P)$ and $NS_1(P)$ must be satisfiable.

Again, let us use the the example path $P = \{FF1, A, D, F, G, FF4\}$ shown in Fig. 1. When propagating logic '0' at launch point $FF1$, we need to justify $NS_0(P) = \{A(0), C(0), F(1), B(1)\}$ in functional mode; when propagating logic '1' at launch point $FF1$, we need to justify $NS_1(P) = \{FF1(1), FF2(1), D(0), C(0)\}$ in functional mode. Since $C(0) \rightarrow FF2(1)$ and $B(1) \rightarrow FF0(1)$, satisfying $NS_0(P)$ implies the existence of illegal state cube $\{FF0(1), FF2(1)\}$. Therefore, P is a timing-independent false path.

3.2 Path-Aware Illegal State Identification

In [12], the authors try to systematically identify all the illegal states in a circuit, which takes non-trivial runtime. For a particular path, however, only those illegal states that are within its fan-in logic cone need to be considered when determining whether it is functionally-sensitizable or not, denoted as *path-relevant illegal states*. Considering the fact that we are mainly interested in critical paths in timing analysis, we propose to adapt [12] and integrate it into our FYP flow as follows.

We first conduct STA on the targeted circuits to find critical paths and record their ending points. Next, for each ending point, we perform structural analysis to trace the relevant flip-flops within its logic fan-in cone. Different from the method in [12], when building the justification schemes, we only consider those state cubes composed of the traced relevant flip-flops. This strategy not only facilitates to save the effort to construct a large number of useless justification schemes, but also automatically avoids to target those non-relevant illegal states. Hence, it improves the efficiency of both illegal state generation and the later false path identification procedures.

3.3 Proposed Examination Procedure

In the proposed method, we first extract illegal states of the circuit according to [12]. Next, we insert *phantom logic AND gates* into the circuit to represent them. As shown in Fig. 2, each phantom gate corresponds to an illegal state by linking its corresponding flip-flop (directly or through an inverter) with a *AND* gate, e.g., illegal state $\{A(1), C(0)\}$ in Fig. 2. This representation has the following advantages:

- we do not need to store illegal states as a separate list (e.g., a large number of independent conjunctive normal form [10]) by naturally integrating them into circuit structure;
- more importantly, by assigning the outputs of the phantom gates to be logic '0's, if a path is sensitizable only with illegal states, it would result in logic conflicts on the phantom gates and we automatically know it is a false path without necessarily generating the vector pair first and checking whether it includes any illegal state cube;

For a given critical path, according to the path sensitization criterion discussed earlier, we conduct two-pass processing to determine whether the path is a timing-independent false path, for justifying $NS_0(P)$ and $NS_1(P)$, respectively. For each pass, we first assign logic values for the circuit nodes according to $NS_0(P)/NS_1(P)$ and logic '0' at all phantom logic gates, while leaving the other circuit nodes to be unknown values. Then, we conduct forward propagation and backward justification to obtain more logic values on those nodes that are initially unknown. If logic conflicts arise during the above logic reasoning process (either at the phantom gates representing illegal states with logic '1' or at a multi-fanout net with contradicting values at its branches), we can conclude the path is a timing-independent false path; otherwise, it is not.

4. FALSE PATH IDENTIFICATION

In this section, we consider the more general problem of identifying as many false paths in the circuit as possible, which is especially important for delay testing. Since the number of the paths is exponential to

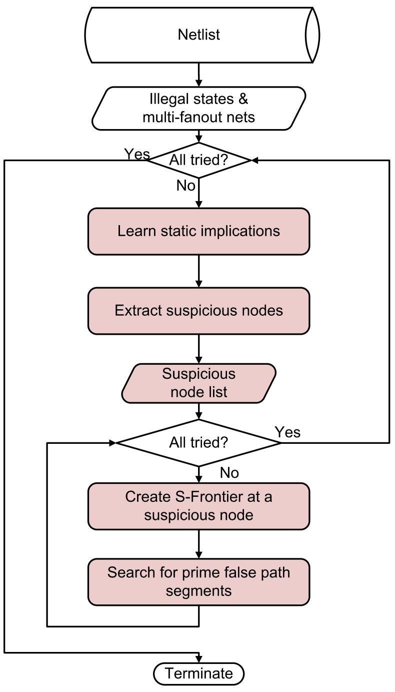
Netlist
Illegal states & multi-fanout nets
All tried?
Yes
No
Learn static implications
Extract suspicious nodes
Suspicious node list
All tried?
Yes
No
Create S-Frontier at a suspicious node
Search for prime false path segments
Terminate

Figure 3: Flowchart for the Proposed Prime False Path Segment Identification Technique

the circuit size, it is apparently impossible to use the technique shown in Section 4 to check path by path. Fortunately, as discussed earlier, timing-independent false paths would imply logic conflicts in the circuit at the phantom gates representing illegal states and/or multi-fanout nets in the combinational logic network. Based on this observation, we propose to identify false paths by targeting at their root causes structures, since the number of inserted phantom gates and multi-fanout nets are much less than the total number of paths in the circuit.

If a *path segment* cannot be activated in functional mode, all paths going through this segment are false paths. In particular, if any section of a false path segment is sensitizable, the false path segment is called a *prime false path segment*. For example, the path segment $\{D,F,G\}$ shown in Fig. 4 is a prime false path segment since sensitizing it implies illegal state cube $\{FF0(1),FF2(1)\}$ while both $\{D,F\}$ and $\{F,G\}$ can be activated in functional mode. Therefore, instead of considering a whole path to be false or not, we target at a path segment in each run because such representation is more efficient.

Based on the above, we propose novel algorithms to systematically identify prime false path segments in a circuit. The basic idea of our approach is to find the minimum path segment P_s whose necessary set-up logic values $NS_1(P_s)$ or $NS_0(P_s)$ imply logic conflicts in the circuit.

4.1 Overall Flow

Fig. 3 presents the overall flow for our systematic prime false path segment identification algorithm.

With given circuit netlist, we first extract illegal states and multi-fanout nets in the circuit. Next, we iteratively target one illegal state or one multi-fanout net in each run. Static implication learning is used to build implications for any possible internal circuit node that can imply values on the targeted illegal state elements or multi-fanout net (detailed in Section 4.2). These implication schemes are then stored in a lookup table as shown in Table 1, e.g., the entry $B(1)$ represents a implication $\{B(1) \rightarrow FF0(1)\}$ for the circuit shown in Fig. 4. For a given path segment, we are now ready to determine whether it is false by quickly checking whether the implications stored in the lookup table lead to any logic conflicts. For example, for path segment $P_s = \{D,F,G\}$, it is false since $D(1)$ and $G(1)$ imply the illegal state cube $\{FF0(1),FF2(1)\}$ according to the implications stored in Table 1.

However, our objective is to identify as many prime false path segments as possible, instead of checking a specific path segment is false or not. Apparently, we cannot afford to consider every circuit node to be the starting point of a false path segment. Fortunately, based on the implication lookup table built earlier, we can extract a set of suspicious nodes as the possible starting point of prime false path segments (detailed in Section 4.3). Then, for each suspicious node, we create a so-called *S-Frontier* to record the path segment we have visited, which contains the following items:

index	A(1)	B(1)	C(0)	D(1)	F(0)	G(1)
context	FF2(1)	FF0(1)	FF2(1)	FF2(1)	FF2(1)	FF0(1)

Table 1: Implication Lookup Table.

- *segment* is used to record all the circuit nodes on the current path segment;
- *launch value* with 1/0 represents that we propagate logic ‘1/0’ at the launch point of the segment stored in *S-Frontier*;
- *implied cube* records all the corresponding implication schemes of *segment* in the implication lookup table;

Then, false segment identification is conducted by propagating *S-Frontiers*, as detailed in Section 4.4. Our program terminates when all the illegal states and multi-fanout nets have been considered.

Again, let us use the example circuit shown in Fig. 4 to illustrate our identification procedure. For illegal state cube $\{FF0(1),FF2(1)\}$, we build its corresponding implication lookup table as shown in Table 1. Consider the suspicious node D with $D(1) \rightarrow FF2(1)$, we create a *S-Frontier* with launch value ‘$1'$ at node D and then propagate it along the path. Accordingly, newly-implied values are continuously added into *implied cube*, and once we reach node G, we obtain {FF0(1),FF2(1)} in the *implied cube* of the updated *S-Frontier* and hence we find a false path segment $\{D,F,G\}$.

4.2 Static Implication Learning

For each targeted illegal state or multi-fanout net, single node implication is utilized to learn which circuit nodes can justify the expected values on them.

Consider the illegal state $\{FF0(1),FF2(1)\}$ for the circuit shown in Fig. 4, we first conduct implication for their inverse values $FF0(0)$ and $FF2(0)$ independently. For example, we can obtain $\{FF0(0) \rightarrow B(0)\}$ since $FF0(0)$ is a controlling value for the *AND* gate. Similarly, we have $\{B(0) \rightarrow G(0)\}$, and hence $\{FF0(0) \rightarrow G(0)\}$ according to the transitivity of implication. By applying counter-positive law, the following implications are stored in the implication lookup table: $\{B(1) \rightarrow FF0(1)\}$ and $\{G(1) \rightarrow FF0(1)\}$. When conducting static implication learning on multi-fanout net, we imply both logic ‘0’ and logic ‘1’ from the targeted multi-fanout net, apply counter-positive law and store the learned information in the same format.

4.3 Suspicious Node Extraction

The starting point of a *S-Frontier* determines whether a false segment can be found and which false segment can be found. Therefore, we need to carefully extract the set of suspicious starting points to create *S-Frontiers*. The selection should satisfy the following requirements: (i). all the possible false segment can be detected; (ii). the selected starting points should be as less as possible.

After static implication learning, we define those nodes that have implications as the *affected nodes*, e.g., nodes A, B, C, D, F and G in Fig. 4. Only affected nodes can serve as the starting point of a prime false segment because the other nodes do not contribute any implications to justify values on the targeted illegal states or multi-fanout nets, and hence they cannot be part of the prime false segment. At the same time, not all affected nodes need to be considered as the starting point of prime false path segments. Take node C as example, which has one implication $\{C(0) \rightarrow FF2(1)\}$. It has two following logic elements D and F. To propagate C(0), we can only generate $D(1)$ and $F(0)$. As can be easily observed in Table 1, however, both $D(1)$ and $F(0)$ imply $FF2(1)$, therefore making $\{C(0) \rightarrow FF2(1)\}$ in fact a redundant implication scheme.

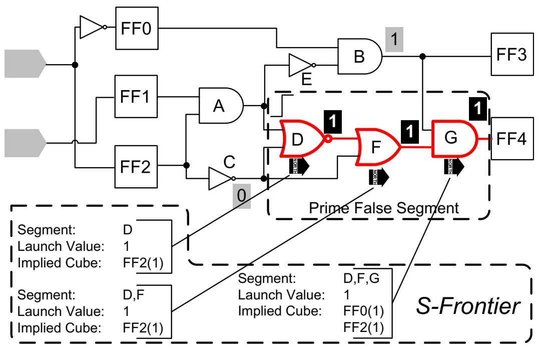

Figure 4: Example to Demonstrate False Path Identification

Benchmark	Flip-flop	Illegal states	Path-Related	Longest paths	PrimeTime	[5]		Proposed		
	#	(#)[12]	Illegal states (#)	(#)	WCD	False path (#)	WCD	False path (#)	WCD	Runtime (s)
s13207	638	82651	1185	5000	13.06	4692	12.89	4986	**12.15**	37.62
s38417	1636	90983	1220	5000	11.42	3582	9.89	4440	**9.76**	8.9
s38584	1426	63558	791	5000	15.07	4380	15.04	4624	**15.03**	25.62
pci	3720	2442	141	5000	5.09	943	5.09	1120	**5.08**	12.083
ethernet	10752	4205	195	5000	9.78	355	9.33	431	**9.02**	236.267
vga_lcd	17265	3096	90	5000	**10.36**	539	**10.36**	793	**10.36**	660.883

Table 2: Experimental Results for Static Timing Analysis.

Based on the above observation, we conduct pre-processing for the affected nodes and we remove those nodes that only contain redundant implication schemes. The rest of the affected nodes are defined as the *suspicious nodes* (e.g. nodes D, F and G), and *S-Frontier* can only be created and propagated from them.

4.4 S-Frontier Propagation

The *S-Frontier* propagation is essentially a breadth-first search process. Firstly, an *S-Frontier* is created at each suspicious node with launch value 0(1) and is propagated along the path. Once an *S-Frontier* reaches a multi-fanout net, it will be split into several copies and delivered to all branches of the multi-fanout net.

After propagating *S-Frontier* to a new node, we first add the new node at the end of *segment* stored in *S-Frontier*, and we add new implications to its *implied cube*, if any. If the on-input of the current node is with non-controlling value, non-controlling value is assigned on the side-inputs and implications for the side-input signals will be also added into *implied cube* of this *S-Frontier*. As shown in Fig. 4, let us consider the propagation of *S-Frontier* from node $F(1)$ to node $G(1)$. Since the on-input of node G is non-controlling value, we also need to assign B with logic '1' and add the implication $\{B(1) \rightarrow FF0(1)\}$ into the *implied cube*. Also, we should check whether the implication schemes with the new node include all the implications with the first node kept in *segment* of the *S-Frontier*. If so, the implication schemes of the first node are redundant and hence we update *S-Frontier* by removing this node to guarantee that the obtained false path segment is a prime one.

Finally, since we may obtain the same prime false path segments starting from different suspicious nodes. Before propagating an *S-Frontier*, we first check whether its starting point is the first node of an existing false path segment. If so, we simply delete this *S-Frontier* as it must have been propagated earlier.

5. EXPERIMENTAL RESULTS

To evaluate the effectiveness of the proposed solution, we perform experiments on several ISCAS'89 and IWLS 2005 benchmark circuits, using a 2GHz PC with 1GB memory.

In our first experiment, we extract a number of critical paths using Synopsys's static timing analyzer *PrimeTime*. As shown in Table 2, we use the proposed method and the implication-based method presented in [5] to examine them. As can be observed, both techniques determine that a large percentage of the reported critical paths are actually false paths. For example, for s13207, by applying our method, only 14 paths out of the the 5000 longest paths are deemed as true paths; while 308 paths are reported to be true with [5]. Consequently, the proposed method is more effective for false path identification.

In terms of timing analysis result, Columns 6, 8 and 10 present the worst case delay (WCD) reported by PrimeTime[1], [5] and the proposed method, respectively. It can be observed that the three methods report the same WCD for several circuits (e.g., *vga_lcd*). This is because the longest paths of these circuits are in fact true paths. For other circuits, the true critical path delay can be reduced after considering the false paths. Due to that our method can detect more false paths, our WCD results is shown to be less pessimistic when compared to [5]. In particular, for $s13207$, the worst case timing delay reported by us is $12.15ns$ while [5] returns $12.89ns$. Our proposed FPI technique is very efficient as indicated by the short runtime, which is an extremely important feature since it can thus be tightly integrated into the inner loop of circuit optimization tools. It should be highlighted that the above runtime includes the time used for path-aware illegal state identification, and as can be seen in Columns 3 and 4, the number of illegal states related to long paths is quite small.

In the second experiment, we present results for our systematic prime false path segment identification algorithm, as shown in Table 3. Column 2 is the number of prime false path segments acquired with the proposed method. We then feed these segments into an ATPG engine built on top of an academic tool *Atalanta* [7] to check whether we can find a vector pair to sensitize them. Column 3 gives the number of segments that the ATPG engine can find a solution to activate them. Conventional method therefore would regard them as true path segments. On the other hand, we also present, in Column 4, the number of segments that the ATPG cannot find a vector pair to activate them. As can be seen in Column 5, in average we can find 202% more false path segments using our proposed method, each of which may correspond to multiple false paths. In particular, for some extreme cases such as *pci*, most of the false segments are treated as true path segments with ATPG-based FPI technique. It should be emphasized that, if we are not able to find a sensitization vector pair for a path segment because ATPG engine aborts computation, we cannot conclude such a path segment is false. The improvement results reported in Table 3 is hence rather conservative. It is also important to note that ATPG-like FPI techniques operate on a 'path-by-path' basis and cannot identify prime false path segments efficiently.

Benchmark	False seg. $\mathcal{F}$ (#)	Justified seg. $\mathcal{J}$ (#)	Unjustified seg. $\mathcal{U}$ (#)	Improvement (%) $I = \frac{\mathcal{J}}{\mathcal{U}} \times 100\%$	Runtime (s)
s13207	80510	11826	68684	17.22	356.32
s38417	68864	10647	58217	18.29	290.52
s38584	37168	5056	32112	15.74	593.65
pci	5052	4644	408	1138.24	1561.48
ethernet	979	153	826	18.52	665.15
vga_lcd	1105	97	1008	9.62	464.58
Average				202.94	

Table 3: Experimental Results for Prime False Path Segments.

6. CONCLUSION

In this paper, we develop novel false path identification techniques by taking illegal states in the circuit into consideration. Experimental results show that our proposed solution is able to find much more timing-independent false paths than existing FPI techniques.

7. ACKNOWLEDGEMENTS

This work was supported in part by a research grant from Cisco Systems, Inc, and in part by a grant N_CUHK417/08 from the NSFC/RGC Joint Research Scheme.

8. REFERENCES

[1] D. Blaauw, R. Panda, and A. Das. Removing User Specified False Paths from Timing Graphs. In *Proc. DAC*, pp. 270–273, 2000.

[2] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De. Parameter Variations and Impact on Circuits and Microarchitecture. In *Proc. DAC*, pp. 338–342, 2003.

[3] K.-T. Cheng and H.-C. Chen. Classification and Identification of Nonrobust Untestable Path Delay Faults. *IEEE Transactions on Computer-Aided Design*, 15(8):845–853, August 1996.

[4] D. H. Du, S. H. Yen, and S. Ghanta. On the General False Path Problem in Timing Analysis. In *Proc. DAC*, pp. 555–560, 1989.

[5] K. Heragu, J. H. Patel, and V. D. Agrawal. Fast Identification of Untestable Delay Faults Using Implications. In *Proc. ICCAD*, pp. 642–647, 1997.

[6] Y. Kukimoto and R. K. Brayton. Timing-Safe False Path Removal for Combinational Modules. In *Proc. ICCAD*, pp. 544–550, 1999.

[7] H. K. Lee and D. S. Ha. On the Generation of Test Patterns for Combinational Circuits. Technical Report 12-93, Dept. of Electrical Eng., Virginia Polytechnic Institute and State University, 1993.

[8] Y.-C. Lin, F. Lu, and K. Cheng. Pseudofunctional Testing. *IEEE Transactions on Computer-Aided Design*, 25(8):1535–1546, August 2006.

[9] J.-J. Liou, A. Krstic, L.-C. Wang, and K.-T. Cheng. False-Path-Aware Statistical Timing Analysis and Efficient Path Selection for Delay Testing and Timing Validation. In *Proc. DAC*, pp. 566–569, 2002.

[10] X. Liu and M. S. Hsiao. A Novel Transition Fault ATPG that Reduces Yield Loss. *IEEE Design & Test of Computers*, 22(6):576–584, Nov.-Dec. 2005.

[11] W. Wu and M. S. Hsiao. Mining Sequential Constraints for Pseudo-Functional Testing. In *Proc. ATS*, pp. 19–24, 2007.

[12] F. Yuan and Q. Xu. On Systematic Illegal State Identification for Pseudo-Functional Testing. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, pp. 702–707, 2009.

[13] J. Zeng, M. Abadir, and J. Abraham. False Timing Path Identification Using ATPG Techniques and Delay-Based Information. In *Proc. DAC*, 2002.

[14] Z. Zhang, S. Reddy, and I. Pomeranz. On Generate Pseudo-Functional Delay Fault Tests for Scan Designs. In *Proc. DFT*, pp. 215–226, 2005.

[1]PrimeTime can report true critical path delay with its own FPI feature.