OBSERVABILITY DRIVEN PATH GENERATION FOR DELAY TEST

A Thesis

by

AVIJIT CHAKRABORTY

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

December 2015

Major Subject: Computer Engineering

ABSTRACT

This research describes an approach for path generation using an observability metric for delay test. K Longest Path Per Gate (KLPG) tests are generated for sequential circuits. A transition launched from a scan flip-flop (SFF) is captured into another SFF during at-speed clock cycles, that is, clock cycles at the rated design speed. The generated path is a ‘longest path’ suitable for delay test. The path generation algorithm then utilizes observability of the fan-out gates in the consecutive, lower-speed clock cycles, known as *coda* cycles, to generate paths ending at a SFF, to capture the transition from the at-speed cycles. For a given clocking scheme defined by the number of coda cycles, if the final flip-flop is not scan-enabled, the path generation algorithm attempts to generate a different path that ends at a SFF, located in a different branch of the circuit fan-out, indicated by lower observability. The paths generated over multiple cycles are sequentially justified using Boolean satisfiability. The observability metric optimizes the path generation in the coda cycles by always attempting to grow the path through the branch with the best observability and never generating a path that ends at a non-scan flip-flop.

The algorithm has been developed in C++. The experiments have been performed on an Intel Core i7 machine with 64GB RAM. Various ISCAS benchmark circuits have been used with various KLPG configurations for code evaluation. Multiple configurations have been used for the experiments. The combinations of the values of *K* [1, 2, 3, 4, 5] and number of *coda cycles* [1, 2, 3] have been used to characterize the implementation. A sublinear rise is run time has been observed with increasing *K* values. The total number of tested paths rise with *K* and falls with number of *coda cycles*, due to the increasing number of constraints on the path, particularly due to the fixed inputs.

ACKNOWLEDGEMENTS

I would like to express my sincerest gratitude and respect for my M.S. committee chair Dr. Duncan M. (Hank) Walker. I would like to thank him for his guidance, encouragement, support and patience throughout the course of the work. I want to thank him for having the faith in me and providing me with all the support and suggestion for all my work.

I am grateful to my M.S. committee Co-Chair Dr. Weiping Shi and committee member Dr. Jiang Hu. I would like to extend my gratefulness for their encouragement and support.

I would also like to thank all the faculty and staff members in both the departments of Electrical and Computer Engineering, and, Computer Science and Engineering for making my experience with Texas A&M University so special.

I want to thank all my friends and colleagues for making this journey so wonderful and enjoyable.

Finally, I would like to thank my parents for their love and support. And a special thanks for my wife for all the support and constant encouragement throughput the whole journey.

TABLE OF CONTENTS

LIST OF FIGURES

Page

LIST OF TABLES

1. INTRODUCTION

1.1 Path Delay Test

Delay test is used to verify the performance of a circuit against its timing specification. The test is designed to test the delay faults which impact the performance of the circuit. This delay can be modeled in various ways. One of the ways is to use a path delay fault model [1] [2] [3]. This model detects both local and distributed faults. A path is generated starting from a Primary Input (PI) or a Pseudo Primary Input (PPI) by traversing through the fan-out cone of a gate and ending at a Primary output (PO) or a Pseudo Primary Output (PPO). The delay test is a scan-based test [4]. The sequential elements in a circuit are connected serially to form a scan chain. These sequential elements provide direct access to the PPI (the output node of a SFF) and PPO (the input node of a SFF). Delay test requires a transition to be launched from a PPI and captured at a PPO. The input of a gate on which such transition is appears is called the on-path input and the remainder of the inputs are called side inputs [5]. In the path delay fault model, the path is said to have a delay fault if the delay of the arrival time of the transition at the capture point exceeds the specified time. The delay of the path is the propagation delay over all the gates on the path. A variety of studies have been done to test delay faults in the circuit [6] [7] [8] [9] [10]. It is intuitive that the path with maximum delay would be the longest path. Hence, to test a delay fault in a circuit, a longest path is generated for the test.

The total number of paths in a circuit has exponential dependency on circuit size. Identification of longest sensitizable paths through each gate or line is extremely difficult. To maintain tractability of a test, K Longest Paths Per Gate (KLPG) are generated in [11]. Tests generated on these paths cover both local delay defects [11] caused by a slow gate and global process variation [12] where more than one path through a gate can be slower than nominal performance.

1.1.1 Delay Test Problem

Figure 1 below illustrates the basic concept of the delay fault. A pair of signal vectors (v1, v2) are applied to the inputs of the circuit (x1, x2, and x3). These vector pairs launch the transition onto the gates. The first vector is called the initialization vector and the second vector is called the test vector. The delay through the individual gates are marked in time units on each gate. The output of the circuit (y) is expected to see a rising transition after seven time units according to the specified propagation delays from individual gates.

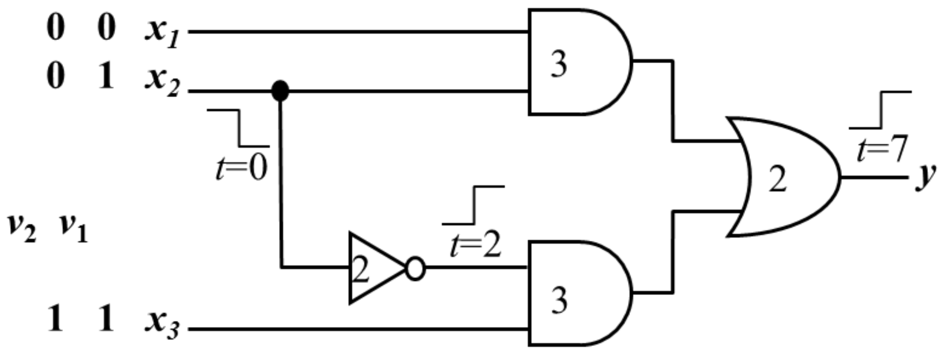


Figure 1. Delay Fault in Combinational Circuit [4]

An additional delay on the path leading to y would delay the rising transition beyond seven time units. Figure 2 illustrates the transitions on input and output of a circuit with specified propagation delay. The shaded region in the figure is called the transition region. The input and output signals are allowed to change states within this region to meet the timing specification of the circuit.

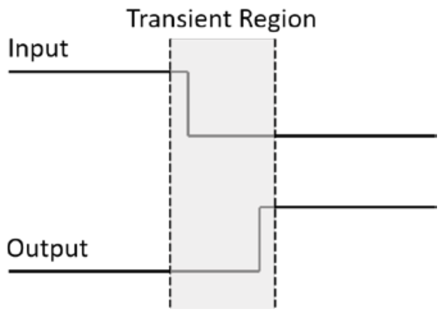


Figure 2. Delay Specification of a Circuit

Any additional delay on the path leading to the output can move the signal outside of the transition region. In case of the circuit in Figure 1, if the timing specification is eight time units and the inverter delay becomes four time units, then the rising transition at the output will occur after nine time units. In such a scenario, the path would experience a delay fault.

A circuit has various paths. The delay of a path depends on the number of gates and the fan-out of such gates. The path with the largest delay is called the critical path. The critical path determines the maximum attainable speed of operation. A delay fault is registered in the circuit when one or more paths experience a delay which is more than one clock cycle.

1.1.2 Path Sensitization

"A path is said to be testable if a rising/falling transition can propagate from the primary input to the primary output associated with the path, under certain sensitization criteria" [13][14][15][16][17][18]. If such sensitization cannot be achieved, the path is said to be untestable or a false path. In order to propagate a transition from one of the inputs of a gate to its output, all other inputs must have *non-controlling* values [4]. Figure 3 illustrates the concept of path sensitization for the path *a-c-d*.

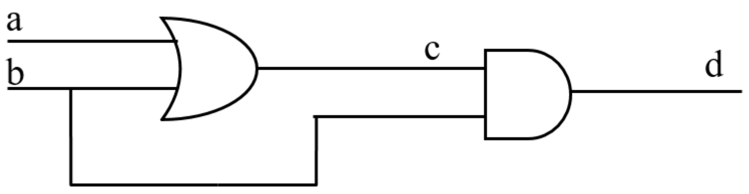


Figure 3. Untestable Path

To propagate a transition launched at node *a*, the side-input *b* of the OR gate needs to be at logic state 0. However, in order to advance the propagation from the output of the OR gate (node *c*), the AND gate side-input (which is also node *b* of the OR gate) needs to be at logic state 1. Clearly, both the gates cannot be sensitized simultaneously. Hence the path *a-c-d* cannot be tested for a delay fault.

1.1.3 Robust and Non-Robust Path Delay Test

A path can be classified as 'robustly testable' or 'non-robustly testable' based on the sensitization criterion. A robust path delay fault test is one in which the delay fault is detected irrespective of any other delay faults that may exist in the circuit. However, a non-robust path delay fault test would detect a fault on a path only in absence of any other delay faults in the circuit.

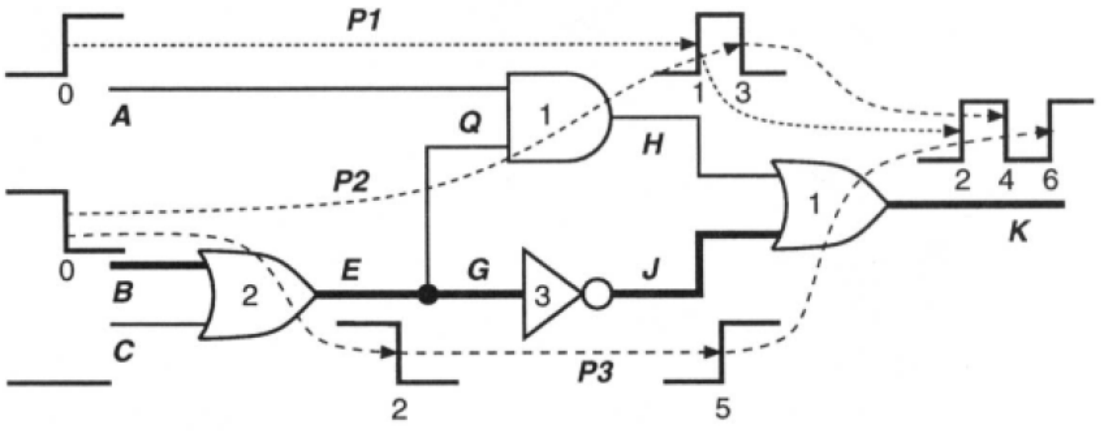


Figure 4. Propagation of transition with path sensitization [5]

Figure 4 illustrates how the presence of faults on a different path in the circuit can mask the fault on the targeted path. All inputs and outputs of the circuit are considered to be synchronized with respect to a clock signal and the clock period T =7 time units. Any path having more than seven time units ofdelay will register a delay fault. The propagation delay values of the individual gates are shown within their individual structures. It can be seen that the path *P3* is the critical path. If all three paths in the circuit (*p1*, *p2*, *p3*) experience time delay of more than seven time units, then the entire waveform will be shifted to the right side and upon observing the output at T=7, a fault will be detected. Now, if *P1* is not faulty and *P2, P3* are faulty, no fault will be observed at T=7 at the output. Hence, the presence of fault *P2* invalidates the fault on the critical path *P3*. Clearly, the fault on *P3* cannot be tested in the presence of other faults in the circuit (a fault on *P2* in this case). For the non-robust test, conditions of static sensitization should be satisfied along with the condition that the test vector pair will produce the required transition at the start of the path under test [4].

1.2 Scan Based Delay Test

Scan design is one of the most widely used approaches in design-for-test. The key feature of scan-based design is the formation of a scan chain. A selected set of the sequential elements in a circuit are connected serially to form this scan chain. Each such element is called a Scan Flip-Flop (SFF). These SFF elements provide direct access to the Pseudo Primary Inputs (PPI: the output node of a SFF) and Pseudo Primary Outputs (PPO: the input node of a SFF). A regular sequential element is converted into a SFF by adding a multiplexer (MUX) at the data input. Figure 5 illustrates the structure of a SFF. It includes a D-type Flip-flop (FF) and a 2:1 MUX.

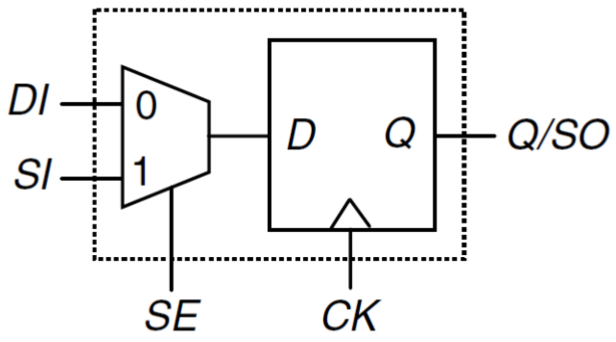


Figure 5. Muxed-D Scan Cell [4]

The scan design utilizes three main signals related to scan operation. The scan input (SI), the scan output (SO) and the scan enable (SE). The SO port of a scan cell is connected to the SI port of the next scan-cell to form the scan chain.

The scan design operates in three modes: *normal mode*, *shift mode* and *capture mode*. First, the scan enable (SE) signal is asserted and the test vector is shifted into the SFFs. This is done at a slow scan clock frequency, typically ten times slower than functional speed. Then the SE is de-asserted and the circuit is put into functional (normal) mode to launch the test vector. One or more at-speed capture cycles (in normal mode) are used to capture the response of the circuit into a SFF. After the at-speed cycles, SE is again asserted to shift the captured response out through the scan chain.

Figure 6 illustrates the scan chain architecture. The SE signal and the MUX structures are not shown explicitly. All the SFFs are clocked with a scan clock (SCLK). The SO port in the figure connects to next set of SI ports.

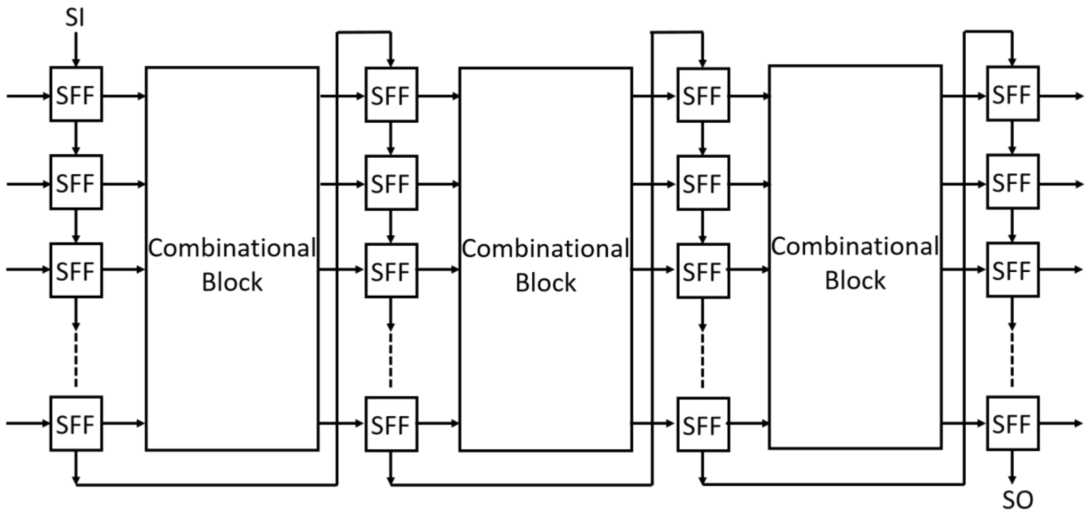


Figure 6. Scan Design Architecture

In a chip, multiple scan chains are formed to maintain the overall time required to load and shift the scan patterns. The following illustrates two different forms of scan design: *Muxed-D Scan Approach* and *Enhanced Scan Approach.*

1.2.1 Muxed-D Scan Approach

Figure 7 shows the example of a Muxed-D Full Scan Design. The DI (data input) port of each of the SFFs is connected to the PPOs of the combinational logic (also known as DUT: Design Under Test). The scan chain is formed according to the description in the previous section. The Q (output) port of each of the SFFs is connected to the DUT as a PPI. The PI (X1, X2, and X3) and PO (Y1, Y2) of the DUT act as the functional signals.

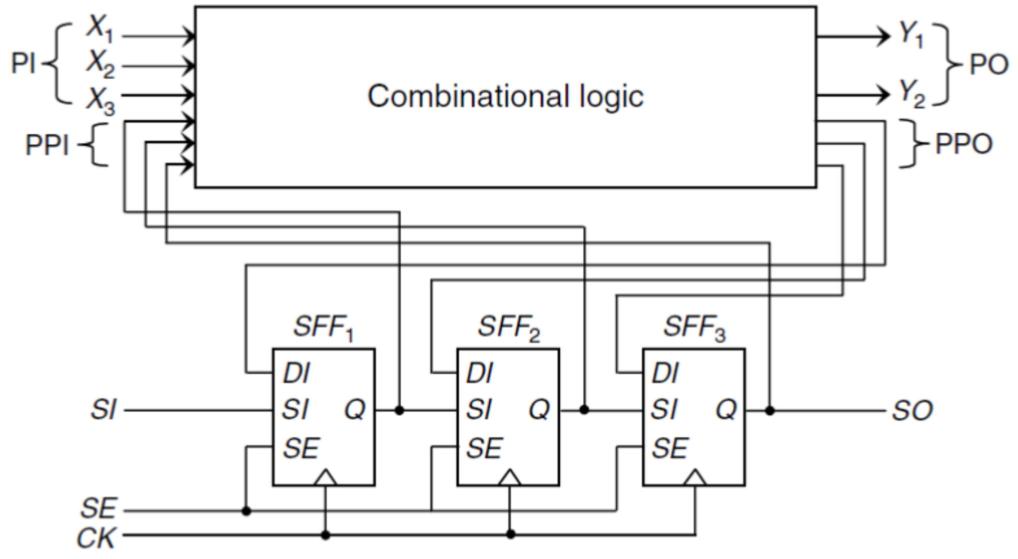


Figure 7. Muxed-D Scan Design [4]

The PI signals are driven by functional upstream logic. The functional output signals (PO) are directly observed as a set of parallel signals while the set of PPO signals are observed through the scan chain. SE is asserted to put the design in scan mode. The SI is shifted into the SFF. After shift is complete, SE is de-asserted and the DUT is put into functional mode. SE is asserted again to capture the response of the DUT.

1.2.2 Enhanced-Scan Design

Delay test utilizes a pair of vectors to launch a transition that is captured at a SFF at functional speed (At-Speed). The nature of the vector pair is kept arbitrary to maximize delay fault detection capability. This is achieved with the help of Enhanced-Scan Design as shown in Figure 8.

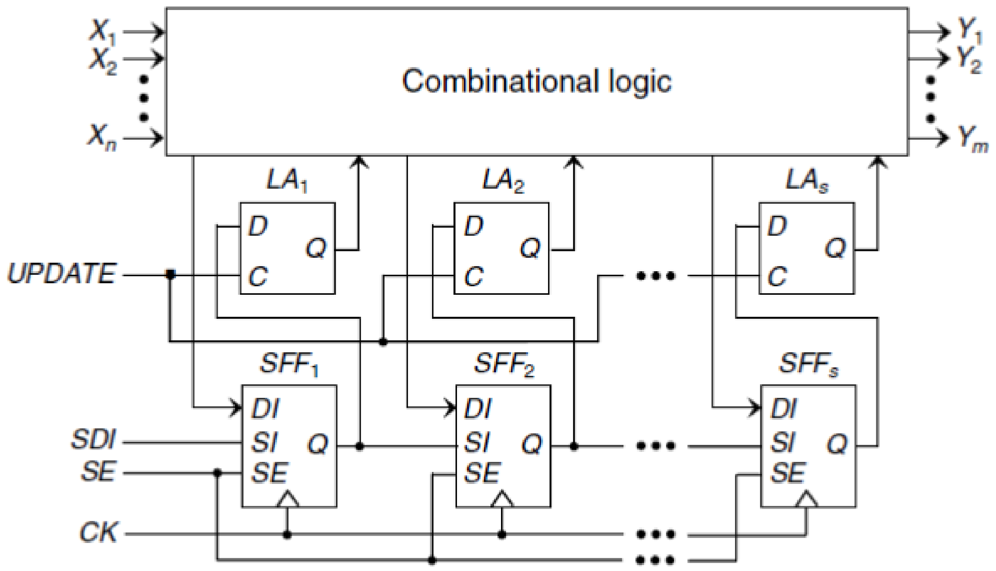


Figure 8. Enhanced-Scan Design [4]

Unlike Muxed-D scan design, two bits of data can be simultaneously applied to the DUT from a SFF. An additional D-Latch is added in order to achieve this feature in the Muxed-D scan design. In order to apply a vector pair (*v1*, *v2*) to the DUT, *v1* vector is first shifted into the SFFs and then stored into the D-Latches. The *UPDATE* signal controls this additional set of latches to store the vector. The second vector *v2* is then shifted into the SFF, keeping *UPDATE* low. After the *v2* vector has been shifted in, *UPDATE* is asserted to change the latch contents from *v1* to *v2,* launching the transition into the DUT. The DUT response is captured in the SFFs.

The main advantage of this architecture is better delay fault coverage by application of arbitrary vector pairs. However, this implementation results in area and delay overhead due to the addition of one extra latch per SFF. The timing between the *UPDATE* and clock signal may also lead to complexities. Another disadvantage is activation of false paths [4] instead of functional data paths, which can cause *overtesting* [4]. Several clocking schemes [4] [19] [20] can be employed to address this disadvantage.

1.3 At-Speed Scan Test Clocking Schemes

At-speed scan architecture enables testing a DUT for it function timing specification. A DUT can consist of multiple clock domains. The clock domains can either be synchronous or asynchronous with respect to each other. Two clock domains are classified as synchronous if the triggering edges of the clocks from these domains can be precisely aligned. In case this alignment is not possible, the clock domains are said to be asynchronous.

There are primarily two clocking schemes for testing inter- and intra-clock domain at-speed faults: (1) Launch-On-Shift (LOS) and (2) Launch-On-Capture (LOC). The following explains the working principle of these two clocking schemes.

1.3.1 Launch on Shift

In this clocking scheme [4], also known as skewed load, the last shift clock pulse is followed immediately by a capture clock pulse to launch the transition and capture the output test response, respectively. The second capture clock pulse is run at the functional targeted frequency, that is, the at-speed frequency. This scheme requires the SE signal to switch very fast, between the launch and the capture clock pulse. This requires the SE signal to be timed at the functional frequency, e.g. a second clock network.

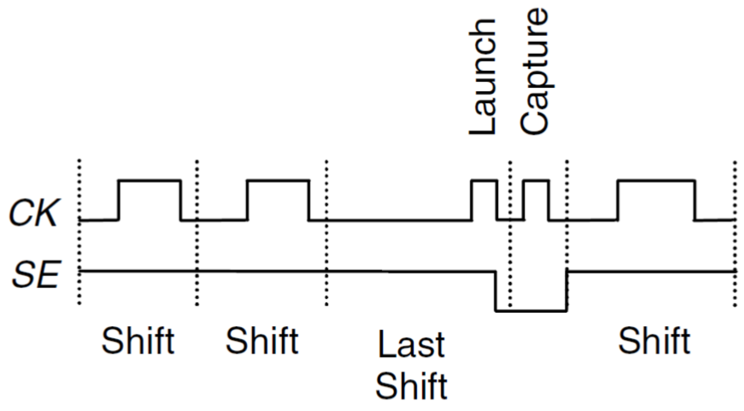


Figure 9. Launch on Shift Clocking Scheme [4]

1.3.2 Launch On Capture

The launch on capture clocking scheme is also known as broadside or double capture mode. It uses two consecutive capture cycles to launch the transition and capture the output response respectively. This scheme does not impose any speed requirement on the SE signal, unlike the LOS scheme. Once the test vector is loaded, SE is de-asserted. Subsequently, the launch and the capture cycles are applied. The LOC scheme typically requires more test vectors and has lower fault coverage compared to the LOS scheme. However, LOC is used more than LOS in high-speed circuits because of relaxed timing on the SE signal.

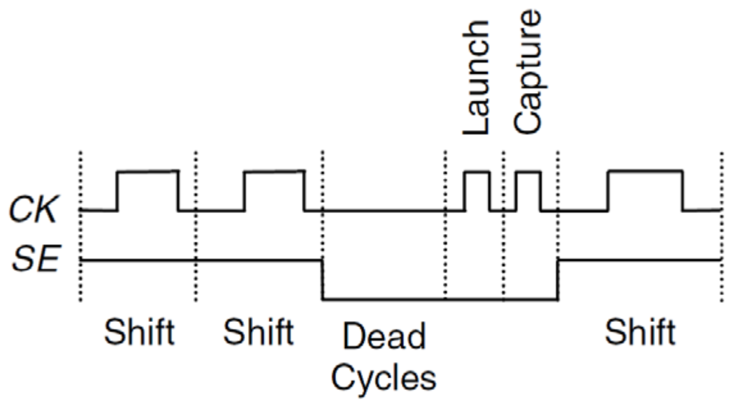


Figure 10. Launch on Capture Clocking Scheme [4]

Several other clocking schemes are available to aid scan designs. Clock domain grouping is utilized to reduce test time and power in scan mode. One-hot clocking and staggered clocking are two such clocking schemes.

1.4 KLPG Algorithm

The KLPG algorithm [21] aims at generating K longest paths through each gate/line in a combinational circuit. A complete path starts at a launch point and ends at a capture point. A launch point is a PI or a PPI and the capture point is a PO or a PPO. The algorithm generates a path by adding one gate at a time, starting at a PI or a PPI. A *partial path* is a path which has originated at the launch point but has not reached the capture point. In the path generation phase of the algorithm, the partial paths are initialized from the launch point. Both rising and falling transition faults are tested through a gate or line. A delay metric called *Esperance* is used to calculate the upper bound of the delay of a partial path. Esperance is calculated as the sum of the delay of the partial path and the PERT delay from its last node to a capture point. “In other words, the Esperance of a partial path is the upper bound of its delay when it becomes a complete path that reaches a capture point. [4]” a flow chart of the algorithm is shown in Figure 11.

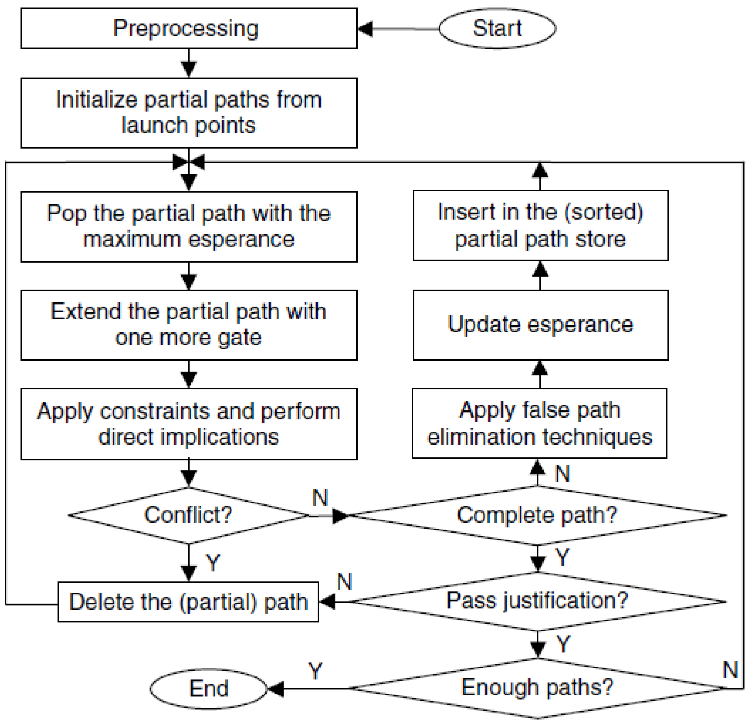

Start
Preprocessing
Initialize partial paths from launch points
Pop the partial path with the maximum esperance
Extend the partial path with one more gate
Apply constraints and perform direct implications
Conflict?
N
Y
Delete the (partial) path
Complete path?
N
Y
Apply false path elimination techniques
Update esperance
Insert in the (sorted) partial path store
Pass justification?
N
Y
Enough paths?
Y
N
End

Figure 11. Flow-Chart of KLPG Algorithm [4]

The KLPG algorithm has three main segments. The path initialization, path growth and path justification. Before a path can be generated, the SCOAP measures [5] of all the gates in the circuit are calculated. The gate connectivity in the circuit is processed and the gates are levelized [5] according to the distance from the PI. The distance of a gate from a PI node is calculated from the maximum of the distance of the gate through its all the input pins. The levelization ensures proper calculation of PERT delay [5], *Controllability* and *Observability* measures. The fan-in and fan-out-cones of individual gates are also calculated in order to derive the SCOAP measures. The controllability is a measure of how easy it is to set the logic level of a node in a circuit to a known state ‘0’ or ‘1’. The observability is a measure of how easy it is to observe the state of a node in the circuit. It is intuitive that the controllability for a node is smaller if it is nearer to a PI or a PPI. Similarly, the observability of a node is smaller if it is nearer to a PO or a PPO. The algorithm calculates the controllability of a gate according to its level in the circuit. The PI and PPI nodes are given a controllability of ‘1’. Controllability of all lower level gates are calculated first before moving to the next level of gates in the circuit. Similarly, the observability is calculated in the opposite order of the level of gates. The PO and PPO nodes are given an observability of ‘0’ and the observability of lower level gates are calculated accordingly.

The next main step is the path generation. One gate is added to the growing path that started at a PI or a PPI, in each iteration. If a gate has multiple fan-outs, the partial path splits at that point to different branches. A pool is maintained to store these partial paths. In each iteration of path growth, the path with maximum Esperance value is extended by adding one more gate to it. This new partial path gets stored again into the path pool and checked for Esperance value before growing it, in the next iteration. Every time a gate is added to the partial path, constraints are added to the inputs of that gate. To ensure propagation of launched transition through the gate, non-controlling values [5] on the side inputs are checked. Direct implication is run to propagate such sensitization constraints throughout the circuit. In case direct implication fails, the partial path is removed from the partial path pool. If the partial path becomes a complete path, Final Justification is run on the path to define the transition vector. This procedure is repeated until sufficient paths are generated through the gate or the line as defined by the *K* value.

Compaction of the test vectors is carried out to reduce the number of test patterns. The compaction can either be *static* or *dynamic*. All independent test vectors are generated before the compaction is carried out in static compaction. The compaction process does not require any circuit analysis, so is fast. In dynamic compaction, compaction is performed as paths are generated. This approach produces fewer test patterns, but at the cost of more memory to store a pattern pool, and justification to check if paths can be compacted into the same pattern. The KLPG algorithm has been implemented in *CodGen* for this work.

1.4.1 Pseudo Functional KLPG

During the time between switching from scan mode to functional mode, when the SE signal is switching, the off-chip currents in the power grid attain a quiescent state. When the at-speed launch and capture cycles are applied, the current demand drastically increases. The off-chip inductance prevents a sudden current increase on the pins, so the current must be supplied by on-chip power grid capacitance. The *dI/dt* phenomenon causes the power grid to experience voltage droop. This causes the chip to perform at a speed lower than the functional specification. This situation can lead to false test failures. Delay test induced droop on the power grid is illustrated in Figure 12.

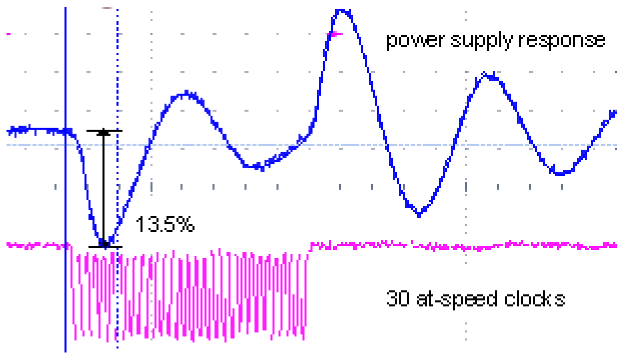


Figure 12. Delay Test Induced DC Droop in Power Supply [22]

The solution to this problem is to apply a number of medium speed *preamble* cycles to ramp the off-chip currents up to functional levels. These preamble cycles filter out most non-functional states, so a test using them is termed a pseudo functional KLPG test (PKLPG) [21]. Figure 13 depicts the generic PKLPG scan clocking scheme.

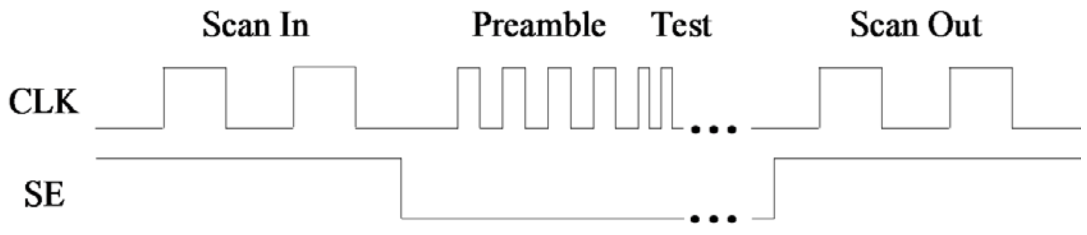


Figure 13. Clock Diagram of Pseudo Functional KLPG test [23]

The SE signal is asserted during scan-in and scan-out operation to shift-in the test pattern and shift-out the test response respectively.

1.5 Boolean Satisfiability

Boolean satisfiability (SAT) has extensive use in the field of Electrical Design Automation (EDA) for circuit verification and testing. SAT solvers typically represent the circuit in Conjunctive Normal Form (CNF) [24]. Techniques such as Boolean Constraint Propagation (BCP) [25] and backtracking with conflict analysis learning have helped developing highly efficient CNF-based SAT solvers.

Boolean or propositional-logic expressions are built using variables, constants and operations such as AND, OR and NOT. The constants are represented in the Boolean form as either *true* or *false*. The satisfiability problem is constructed by setting *truth assignment* (assignment of ‘0’ or ‘1’ value to each of the variables) to the variables that make the value of the function ‘1’. Hence the main goal is to derive the assignment of variables that makes the functional value a *true* (‘1’). The problem of satisfying a CNF formula using SAT is NP-complete, so heuristics are used to speed up the solver.

The CNF for a function is expressed in terms of individual *clauses* which are combined with an AND operation. Each clause is represented as an OR of *literals.* A literal can either be a variable or the negation of that variable. For example, the operation AND can be expressed as $Z = X \cdot Y$. The CNF representation for Z will be $(\sim Z+X)(\sim Z+Y)(\sim X+\sim Y+Z)$. The solution space will be the values of X, Y and Z such that the formulated equation has a value of ‘1’. The clause with 2 variables is called *2CNF* and can be solved in polynomial time. The clause with 3 variables is called *3CNF* and it is a NP-complete problem. An XOR relationship between variables in a clause is handled as a system of linear equations and solved in cubic time.

Application of SAT for test vector generation in ATPG experiences the difficulty of incorporating real delay values. A mix of structural and functional test approach has been implemented in [26] to alleviate this problem. In this implementation, the paths are generated with a structural approach and SAT is used for the path justification. A variety of approaches have been discussed in the literature to speed up the SAT solver. Dynamic SAT Solving (DSS) structural information of a circuit is used by the SAT engine to improve solution time [26]. CNF based SAT solver has an inherent issue of loss of circuit structural information during the CNF formation. Structural information like direction of gates and Circuit Observability Don't Cares (Cir-ODC) are potentially useful in the solution process. The approach of utilizing the Cir-ODCs is explained in [26]. In this approach, the Cir-ODCs are identified prior to the SAT solving step. The decision heuristics, BCP and the conflict driven procedures are adjusted according to this information.

1.5.1 Use of SAT in CodGen

MiniSAT [27] is an open source SAT solver used in CodGen. SAT is used in the final justification and dynamic compaction stages of the KLPG algorithm. Final justification is run on a complete path for all the values assigned to the gates (*necessary assignments*). Since the LOC clocking scheme utilizes a vector pair for launching the transition, two variables are used for SAT solving in two time frames. For pseudo functional KLPG, more than two variables are required for the solution, due to the separate input values on each clock cycle. In the case of fixed PIs, as is typical in low-cost testers, only one Boolean variable is required across all the time frames.

1.6 Structure of the Thesis

In this thesis we propose an Observability driven path generation methodology for at-speed delay test. The proposed approach is useful in path growth beyond the at-speed cycles for a design in which not all flip-flops are part of the scan chain. The observability metric driven path growth ensures the generation of a *complete path* ending at a SFF.

The thesis is organized as following. In section 2, we present the motivation behind the work. Section 3 describes the implementation of the observability based path generation strategy in a modified KLPG algorithm. Section 4 discusses the results of the experiments on various ISCAS89 benchmark circuits. Section 5 concludes the research.

2. MOTIVATION

2.1 Scan Architecture for SOC

Modern microprocessors embed billions of transistors on a single die. The methodology and complexity of testing a design of such scale is mindboggling. Silicon real estate required to carry out such complex test of circuits can be a sizable percentage of the die area [28]. Scan design is one of the most preferred techniques for on-chip testing. The sequential elements are converted into SFF. The SFF cells are connected serially to form scan chains. Several scan chains can be formed on a chip to limit the total number of SFF on a single chain. The amount of time required to load the chain with the test vector and shifting the response through the entire length of the chain can take up to thousands of functional clock cycles. Clearly, shorter and fewer scan chains on a chip is desirable. It saves area and test time which directly impacts the bottom line for a product.

At-speed delay test utilizes a pair of SFF. A transition is launched from one and the response of the DUT is captured in the other. The number of at-speed cycles and timing relation of SE signal with respect to test mode enable signal of the chip is defined by the test architecture. To ensure a proper capture of the test response from the DUT into a SFF, all the SFF should be part of the scan chain and should accommodate the total number of at-speed cycles before SE is asserted to shift out the test response. If some of the sequential elements in the design are not part of the scan chain, the test response may not get captured to be shifted out on the SO signal. This results in fault coverage loss.

2.2 Path Generation in CodGen

The current *CodGen* pseudo functional KLPG system generates longest paths over multiple at-speed cycles. *Esperance* of the partial path determines the branch of the fan-out cone through which the path grows. This metric is applicable for the at-speed cycle to ensure that the path is *longest*. The designation of a path as *complete* is determined by the termination point of the path at the input port of a SFF. (Low cost testers do not capture PO outputs). The current implementation assumes that all sequential elements (FF) in the design are SFFs. Hence, a path is designated as *complete*, even if the sequential element at the end of the path is not a SFF. In reality, this generated path would be a *false path*, since the test response captured in this FF would not get shifted out through the scan chain.

2.2.1 Clocking Scheme

In order to move the response captured in a FF to a SFF, a number of lower than at-speed cycles are added after the capture cycles before asserting the SE signal. These cycles are known as the *coda* cycles. The timing of these cycles is such that circuit delay does not need to be considered in these cycles. Figure 14 illustrates the clocking scheme for such implementation.

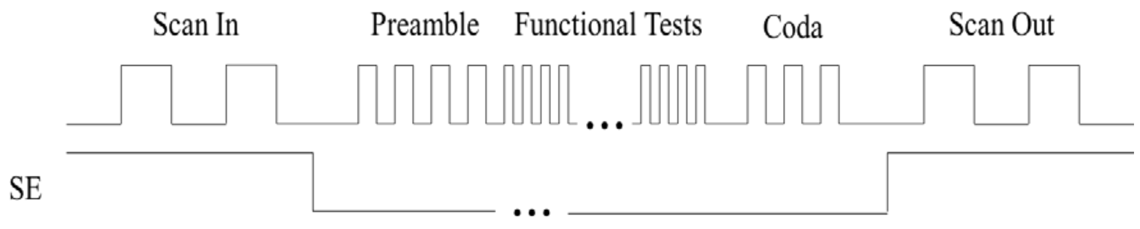


Figure 14. Inclusion of Coda Cycles for At-Speed Delay Test

The *coda* cycles ensure that the transition from the at-speed cycle can still be captured into a SFF in the event that the FF capturing the transition in the last at-speed cycle is not part of the scan chain. A minimum amount of scan infrastructure needs to be in place to ensure the coverage goal. For a given clocking scheme in terms of number of coda cycles, if the design incorporates less than the critical level of scan elements, the at-speed transition cannot be captured with the help of a designated number of coda cycles.

2.2.2 Observability Driven Path Generation in Coda Cycles

The path generated in the at-speed cycle needs to be *longest* for delay test. However, in the case of a *coda* cycle, the path starting at the first capture flip-flop, need not be a longest one. The main objective of this path is to transfer the captured value to a SFF. This is implemented using observability of a gate as a metric in growing the path in the coda cycle. Observability based path generation ensures that the most observable path is generated first in the coda cycle. This path will be the most compatible with the necessary assignments of the tested path. *Esperance* based path growth does not differentiate between a scan and a non-scan flip-flop. Hence, it can report a path in a coda cycle that ends at a non-scan element, which will essentially be a false path. In the same scenario, the observability metric determines the path to be un-observable by assigning infinite observability (a lower observability value means the line is more observable) for a non-scan flip-flop, forcing the search process to consider an alternate path.

2.3 Related Prior Work

The KLPG algorithm is described in detail in [11]. The algorithm generates *K* Longest Paths per Gate for both combinational and sequential circuits. The extension of the KLPG algorithm over multiple cycles is described in [23]. The process of longest path generation is replicated across all the clock cycles. *Esperance* is used to determine the longest path in all of these cycles. Test pattern generation using a SAT engine for justification is described in [29]. The work described here builds on top of these prior ATPG tools. The overall name given to these tools is *CodGen*, since it targets combined local/global delay defects.

Critical path tracing methodology is discussed in [30]. A hybrid scan-based technique has been proposed in [31]. It uses controllability measure to control a subset of the scan cells using either LOS or LOC clocking scheme. The paper enumerates the saving in scan enable design effort. However, it does not evaluate the path generation using the proposed scheme.

3. IMPLEMENTATION

3.1 Path Generation Strategy in KLPG Algorithm

The KLPG algorithm implemented in *CodGen* is capable of generating longest paths over multiple at-speed cycles [23]. It utilizes an *Esperance* metric to ensure that the generated path is *longest* for delay test. The current implementation incorporates preamble cycles followed by multiple at-speed cycles. All the sequential elements in the circuit are assumed to be SFF. Hence it does not distinguish between a SFF and a regular FF. We have extended the clocking scheme to incorporate the *coda* cycles. The enhancement also incorporates the *observability* metric for path generation in the *coda* cycles. Hence, the different clock modes are distinguished with respect to the path generation metric. The clocking modes implemented in *CodGen* are illustrated in Figure 14.

The sequential elements have been extended with a field to differentiate the cell to be a SFF or a regular FF. The implementation ensures that the path generated in the at-speed cycle remains the longest but the path growth in consecutive coda cycles is driven by the observability metric rather than the *Esperance* value. The code also ensures that no path is generated if the last capture flop on the path is a non-scan flip-flop. Any intermediate non-scan flip-flop is considered as a regular flip-flop with infinite observability but does not result in dropping of the path from the partial path pool. The algorithm works for as many coda cycles as specified. A search tree for the path is shown in Figure 15.

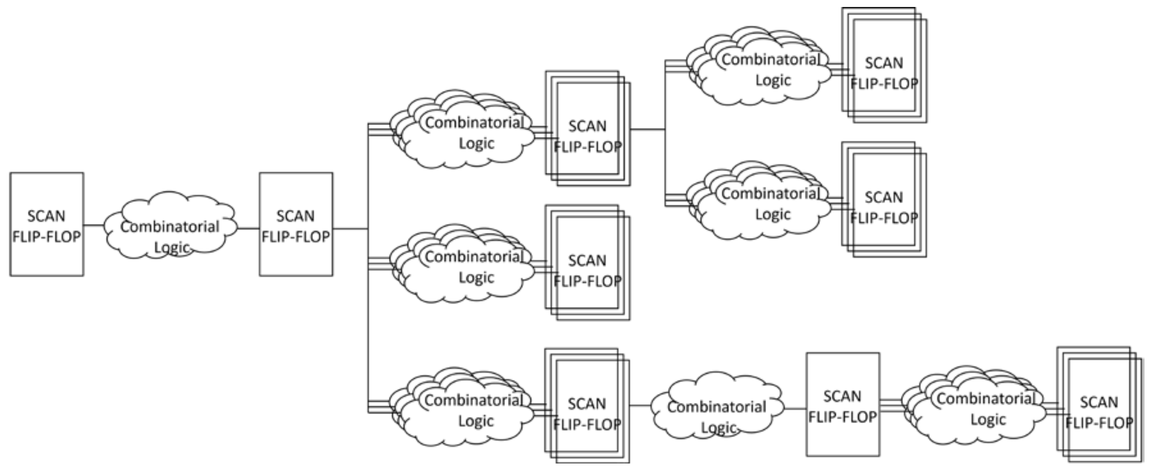


Figure 15. Path Generation using Esperance and Observability Metric in Sequential Circuit

One of sequential elements is marked as a non-scan regular FF. All the gates in the fan-in cone of this FF see infinite observability coming from the PPO. In case of a generated path for a given number of *coda* cycles, if this FF acts as the terminating PPO, the path is discarded instead of being reported as a complete path. If this FF is one of the intermediate sequential elements on the path, the corresponding branch is considered at the last decision point for partial path growth, only after all other branches from its fan-in gate has already been tried for path growth.

The enhanced implementation works with as many *coda* cycles as specified in the clocking method. The path generation step is followed by justification in the same manner it is carried out for the at-speed cycles.

The following section elaborates on the steps involved in the path generation using the KLPG algorithm.

3.2 *CodGen* Functions for Gate Processing

3.2.1 Circuit Verilog Parsing

CodGen uses a Verilog netlist of the circuit to create an in-code circuit representation in the form of the connections between the gates. The logical gates (AND, OR, NAND, NOR, XOR, XNOR, NOT, BUFF, MUX, etc.) are defined in the *Gate* class. The TruthTable of each of these gates describes the logic operation for each of these gates. The sequential elements are described in a separate Verilog file which groups the FFs in terms of the scan chain. An additional file containing scan cell attributes has been implemented in this work to designate the FF cell as a SFF. This *scan-cell-attribute* file is parsed along with the Verilog files and the corresponding field in the *gate* object in the code is updated accordingly. This field (SC) is utilized by the observability calculation program to propagate the observability in the fan-in cone of any object of the *Gate* class.

Each of the gates in the circuit is given a *GateID*. The PI, PPI, PO and the PPO nodes are also treated as gates in the *Gate* class to streamline the path generation. The net connected to the output of the gate acts as the *NETID*. This *NETID*, in turn, gets stored as input net of one or more gates in the fan-out cone of this gate. The gates are *levelized* [5] according to the distance of the gate from the input and output nodes (both primary and pseudo primary type) of the circuit. The *levelization* ensures correct SCOAP measure [5] calculation in the following stages.

3.2.2 Gate Delay Assignment

CodGen can assign realistic delay for a given gate in the circuit for both rise and fall transition at the output node if a SDF delay file is provided as a part of the input files to the code. In absence of such delay file, a unit-delay-model is utilized to assign gate delays.

3.2.3 Fan-In/Fan-Out Cone

Fan-in and fan-out cones of each gate are calculated in order to create the in-code circuit structure. The search space in the KLPG algorithm in the form of the fan-in and fan-out cones is shown in Figure 17.

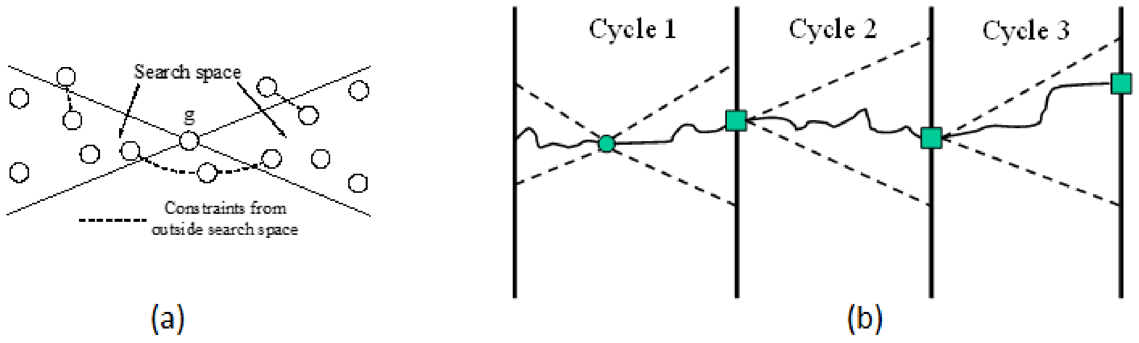


Figure 16. KLPG search space [4][23]

The fan-in and fan-out paths are the search space of the gate *g* in the figure. The line through the cones are designated as *on path* [5]. The paths outside this search space provide *side input* [5] values.

The fan-in and fan-out cone of gates are calculated in the same manner in case of multiple at-speed and coda cycles. This is illustrated in Figure 17(b). Proper *side input* assignments are necessary over multiple *time-frames* [4] for correct propagation of the transition launched at the PPI.

3.2.4 Circuit Initialization

Controllability and Observability of the gates are calculated in this step. Controllability is defined as ease of controlling a node in a circuit to a logic level of ‘0’ and ‘1’. The combinational controllability metrics are denoted as *CC0 and CC1* accordingly. The PI and the PPI nodes are assigned *CC0* and *CC1* values of ‘1’ since these nodes can be directly controlled. The *CC0* and *CC1* metrics of all subsequent gates, in the level order, in the circuit are then defined according to the logic of the gate. It should be noted that the controllability measures of the input of a gate at a given level is measured only when all the lower level gates have already been assigned with the controllability values for all of their input nodes. This ensures a correct calculation of such metric. In case of observability metric, the calculation starts at a PO or a PPO. Observability of ‘0’ is assigned to these nodes. The observability function implemented as an enhancement in this work stores the combinational observability (*CO*) of both input and output nodes of a gate. The output *CO* is used for the gate observability while the input node *CO* values are used to define the same for a gate in the fan-in cone which has more than one fan-out. The *CO* of a gate with more than one fan-out is defined as the minimum of the *CO* values from all of its fan-out cone gate inputs. The observability function assigns infinite observability for a PPO of a non-scan FF. The infinite value is accordingly propagated to all the fan-in cone gates of the non-scan FF. Figure 16 illustrates the SCOAP measures of a representative circuit.

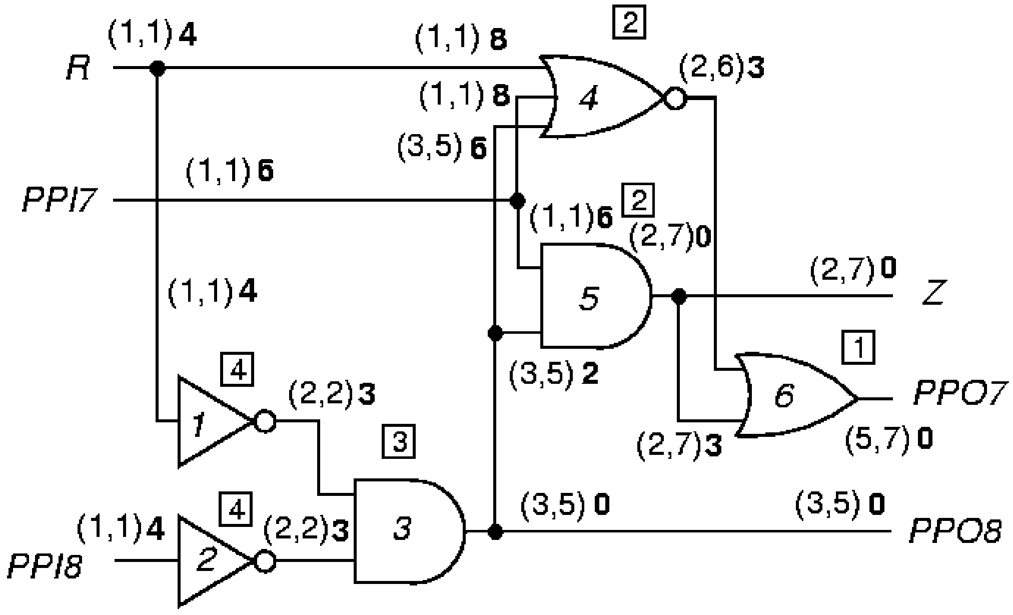


Figure 17. SCOAP Metric Calculation [5]

The level of the gates from PPOs is represented as numbers inside a square. Each gate has an identifying number in it. The notation (CC0,CC1)CO is used to list CC and CO values for each line in the circuit.

3.3 Path Generation Aiming the Scan Sequential Element

Previous versions [11] [23] of the KLPG algorithm attempted to generate paths through all the gates. We consider paths starting at a SFF for at-speed scan test. Hence the path generation aims at generating a path starting from a SFF. The function

AimingPathGen generates a partial path pool which keeps the record of the path which gets expanded in an iterative manner by adding one gate each time to the partial path, from the fan-out cone of the last gate of the partial path. Both the rising and falling transitions through a gate are used to grow a path. The total number of iterations for each transition from a SFF is given by $K{\cdot}j$ wherein K is the number of paths per gate targeted for path generation and j is the total number of fan-out stems of the SFF.

The partial path pool is always sorted according to the *Esperance* value of the partial paths. It ensures that each iteration always uses the partial path that is most eligible to become a *longest complete* path. When a PPO gate is added to one of such partial paths, it becomes a *complete* path and it is removed from the partial path pool. The path is reported after successful final justification.

3.4 Time-frame Expansion

The test pattern generation is implemented in *CodGen* uses a time-frame expansion methodology. A Boolean gate-level model [4] of the circuit is generated in this method to generate the tests using a combinational ATPG method. The combinational part of the circuit is expanded in time by using the logic twice for a pair of time frames. The vector pair of the delay test, is applied to the expanded circuit. The two time frames are denoted as *frame -1* and *frame 0*. The circuit block corresponding to *frame 0* receives its input from the response of *frame -1*. Figure 18 illustrates the concept of using time frame expansion used for LOC scheme.

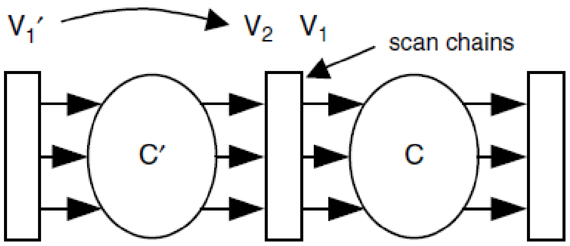


Figure 18. Time-frame Expansion for Delay Test [4]

The objective of this step of the path generation is to ensure that an intended transition can be propagated through a gate, i.e. required logic values can be assigned for path sensitization. For a rising transition, the vector in *frame -1* will be ‘0’ and it will be ‘1’ for *frame 0*. It will be opposite in case of a falling transition.

If the last gate on the partial path has more than one fan-out, the path splits into branches. Constraints are applied on the last added gates to ensure the propagation of the transition obtained from the previous step described above. Direct implication [4] is then performed in a recursive manner on each of the gates in the fan-in and fan-out cone. Whenever, the logic state of the output node of a gate changes due to the application of the input transition, it needs to be propagated to the downstream logic. The side inputs of the gate need to be at non-controlling state as well.

The direct implication can fail whenever the required logic values cannot be assigned to a gate. Conflict can arise in a circuit where the assignment of a non-controlling value on one gate may disable a different gate sharing one of the same inputs. For non-robust sensitization, the side inputs need to have non-controlling values. In case of robust sensitization criterion, the side inputs need to remain at the non-controlling value if the on-path signal makes a transition to a controlling value. Figure 19 illustrates a scenario of conflict during path sensitization.

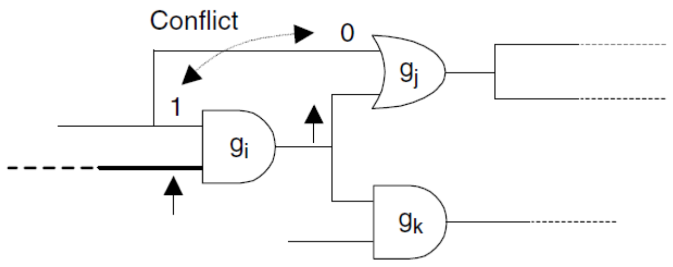


Figure 19. Conflict in Logic State Assignment [4]

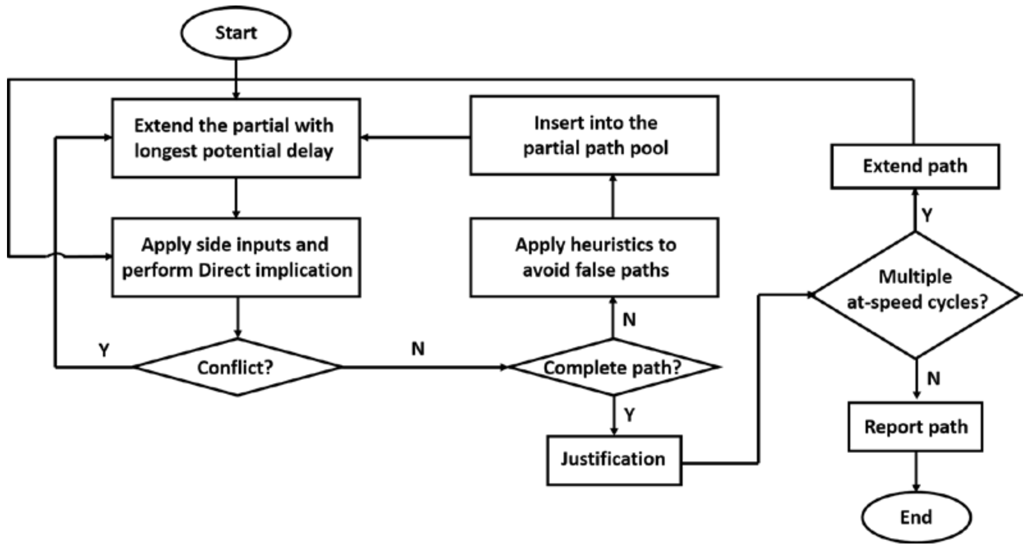
The gate g_i in the figure represents the last gate on the partial path. The gate g_j is in the fan-out cone of g_i. In order to propagate the transition through the gate g_i, the side input needs to be at logic '1'. However, assigning this logic value to g_i will block propagation of transition through the gate g_j. In the event of a failed direct implication, the path growth is inhibited through the gate. A variety of heuristics [4] can be applied to trim the search space for any other path generation through the same line.

3.5 Final Justification

Final justification is performed before a path can be reported. In case of multiple at-speed and *coda* cycles, the justification step is carried out on each cycle before the path is extended in the consecutive cycles. If the justification is run only at the end, and justification fails for a gate which is part of the first capture cycle, then the path generation has to be rerun for all the subsequent cycles. This would lead to unnecessary and additional iterations for the same lines with clean justification in the later cycles.

4. CODA CYCLE PATH GROWTH

The path growth in coda cycles is based on the observability metric. If the last gate in the partial path has more than one fan-out, the gate with minimum observability at the input node gets added to the partial path pool. The partial path pool is ordered according to the observability overhead. This reordering of the partial path pool ensures that the path generated in the *coda* cycles is most observable. The flowchart in Figure 20 depicts the path generation algorithm for multi-cycle path generation.



Start
Extend the partial with longest potential delay
Insert into the partial path pool
Extend path
Apply side inputs and perform Direct implication
Apply heuristics to avoid false paths
Multiple at-speed cycles?
Y
N
Conflict?
Complete path?
Justification
Report path
End

Figure 20. Path Generation Algorithm for Multiple At-Speed Cycles

The *coda* cycle path generation utilizes the same approach for the path growth as shown in Figure 20. It involves an additional step of reordering the partial path pool according to the observability metric prior to extending the path in each iteration. The total number of at-speed and *coda* cycles can be specified in *CodGen* along with the *K* value and sensitization criterion to generate paths for delay test for such a clock configuration.

Figure 21 shows a modified version of the ISCAS89 S27 circuit. The S27 circuit has been modified to incorporate multiple at-speed and *coda* cycle paths for code debugging and algorithm illustration. The SFF elements are denoted with ‘SC’ and the regular FF is denoted with ‘NS’.

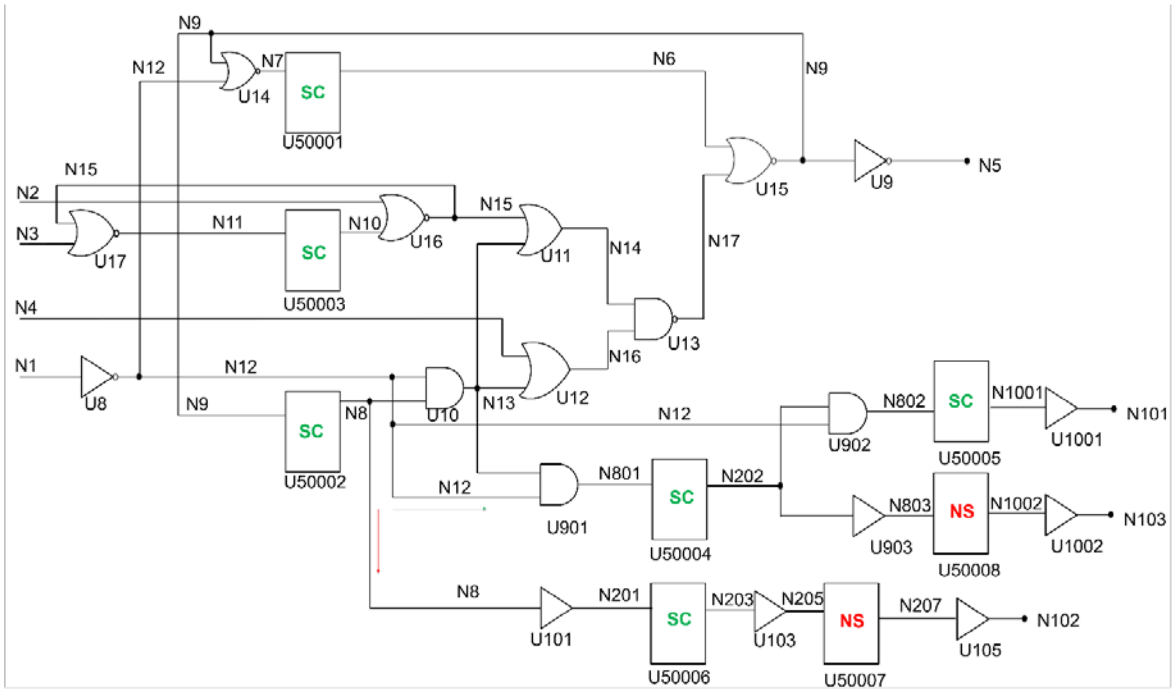


Figure 21. Modified ISCAS89 S27 Benchmark Circuit

Considering one at-speed cycle and one *coda* cycle, the paths starting at U50001 SFF are:

Path 1: U50001—U15—U50002—U10—U901—U50004

Path 2: U50001—U15—U50002—U10—U50006

The single at-speed cycle path starts at U50001 and ends at U50002. The fan-out of the SFF U50002 consists of two gates: U10 and U101. The *Esperance* of Path 1 is greater than that of the Path 2. However, the observability overhead through gate U101 is lower. Hence, in the *coda* cycle, Path 2 is generated ahead of Path 1.

For one at-speed cycle and two *coda* cycles, the paths starting at U50001 SFF are following:

Path 1: U50001—U15—U50002—U10—U901—*U50004—U902—U50005*

Path 2: U50001—U15—U50002—U10—U901—*U50004—U903—U50006*

Path 3: U50001—U15—U50002—U10—U50006—U103—U50007

This configuration models the presence of non-scan elements during the path growth. The SFF U50004 has a fan-out of 2. The *coda* cycle path originating from this SFF follows the same observability driven ordering as described in the previous configuration with one *coda* cycle. As a result, Path 1 is generated before Path 2 is considered. However, the FF U50006 has the 'NS' attribute, making it a regular non-scan FF. Since, this is the only terminating FF in the current branch of the circuit, Path 2 is not grown to this FF. Since U50007 is also a non-scan FF, Path 3 is not generated either.

For the same clock configuration as above, if gate U50004 is made 'NS' and both U50005 and U50006 are made 'SC' , then both of the paths (Path 1 and Path 2) would be reported. Having a non-scan element as an intermediate FF does not impact capturing the transition at the terminating SFF. If U50007 is kept as a non-scan cell, only two paths are generated. If U50007 is made SFF, then Path 3 gets generated first due to lower observability (since there are no non-scan cells on this path, the observability metric of the corresponding branch is reported to be lower than the branch through the gate U901 which *sees* infinite observability propagated from the non-scan FF U50004.

5. RESULTS

The modified KLPG algorithm is implemented in C++ on an Intel Core i7 machine with 64GB memory. Experiments are carried out on ISCAS89 benchmarks and industrial circuits. The code has been run with different *K* values and coda cycles. The sequential cells has been configured as scan and non-scan cells to ensure the validity of the code in terms of handling different observability overhead. Circuit inputs (PIs) are fixed during each test pattern (resembling a low cost tester). The code has been validated to ensure distinction between the *Esperance* and Observability driven path expansion in at-speed and coda cycles respectively. The partial path pool has been thoroughly validated to ensure correct ordering based on *Esperance* and observability measures. Different sequential elements have been configured as scan or non-scan elements to ensure correct behavior of discarding or growing a partial path in a given clock scheme.

5.1 *CodGen* Run Time and Path Generation Results

Figure 22 shows the trend for run time with increasing numbers of paths per gate *K* and number of coda cycles (CYC) for the benchmark circuit *s1488*. As expected, run time rises sublinearly with *K*. Run time does not necessarily rise with CYC due to the changing number of tested paths. Figure 23 shows that the number of tested paths rises with *K*, as expected, and falls with CYC, due to the increasing number of constraints on the path, particularly due to the fixed primary inputs.

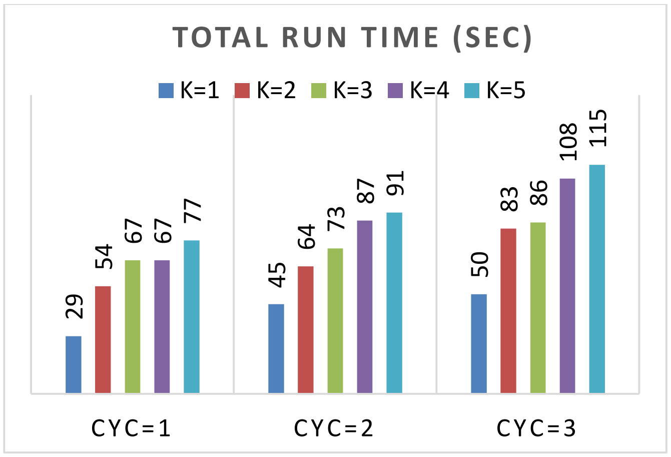


Figure 22. Run Time for s1488

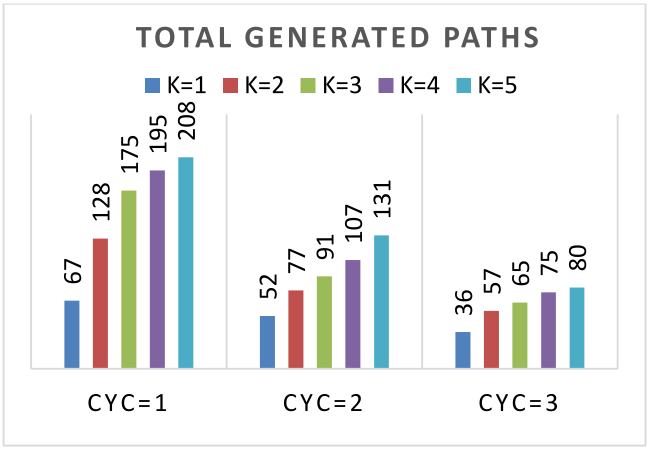


Figure 23. Generated Paths for s1488

A similar trend is observed for the other benchmark circuits as well (see Appendix I). It should be noted that the increase in the number of paths generated with increasing value of *K* gradually decreases for all the cycles. This observation is captured in Table 1.

Table 1. Relative Increase in Number of Paths with Increasing *K* Value

K	Relative Increase in No. of Paths (K_n/K_{n-1})		
	CYC=1	CYC=2	CYC=3
2	1.9	1.5	1.6
3	1.4	1.2	1.2
4	1.1	1.2	1.1
5	1.1	1.1	1.1

Figure 24 illustrates the trend captured in Table 1. Although the variation in the increase in the number of paths as a function of increase in the value of *K* depends on the structure of the circuit, the trend observed over various circuits and number of cycles is the same (see Appendix I).

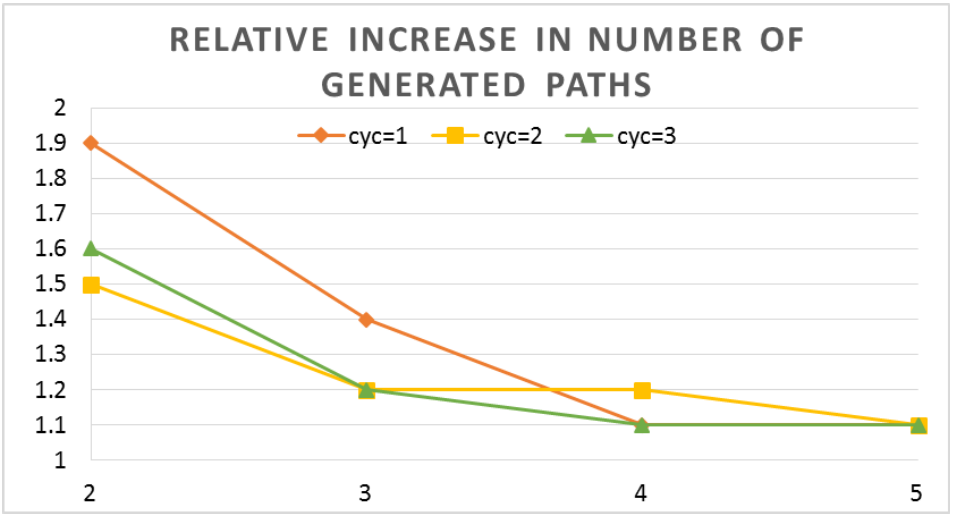


Figure 24. Relative Increase in Number of Paths with Increasing K Value

The total number of paths that can be generated through a gate primarily depends on its fan-outs (apart from the fact that the direct implication has to pass). For a low value of *K*, not all such paths get generated in the KLPG algorithm. The path limit defined by the *K* value gets exceeded before all the paths can be explored. Clearly, increasing the value of *K* helps generating more paths in such a scenario. However, it is intuitive to realize that the total number of generated paths would not increase beyond a certain number even with a high value of *K* if all possible fan-out cones have already been explored. In presence of side input constraints, the increase in the number of generated paths can be lower than expected. The experiments run with *CodGen* show that not enough benefit is achieved in order to generate more paths per gate for values of *K* greater than 5.

Table 2 shows the total run time and total paths generated for a configuration of *K*=1 and CYC=1 for various ISCAS89 benchmark circuits. The longest path length is also captured in the table.

Table 2. CodGen Run for Configuration: K=1, CYC=1

CodGen Run Configuration: K=1 CYC=1					
Benchmark Circuit	# Gates	# FF	# Paths	Longest Path Length	Run Time (s)
s1494	661	6	55	24	29
s1488	667	6	67	26	29
s1423	748	74	74	39	185
s5378	2993	179	235	25	97
s9234	5844	211	143	52	652
s13207	8651	639	224	62	458
s15850	10833	534	443	65	1206
s38584	22142	2426	1294	71	3678
s38417	23843	2636	957	51	10896

Figure 25 shows the trend in the run time as a function of total number of gates in the design. It can be seen that the run time increases at a linear rate until *s38584*. The run time for *s38417* is almost 3X that of *s38584*. This abrupt change in the run time can be understood from the number of *Aiming* trials for each of these runs. For *s38584*, the *CodGen* run tries to extend a partial path a total of 863,897 times. In case of *s38417*, this number is 4,014,779. A large percentage of the total run time is spent in the path generation step of *CodGen*. Hence, a larger number of path extension trials directly impacts the total run time. The structure of the circuit has a key role to play in defining the partial path growth and can lead to non-linear dependency for run time and total number of testable paths generated.

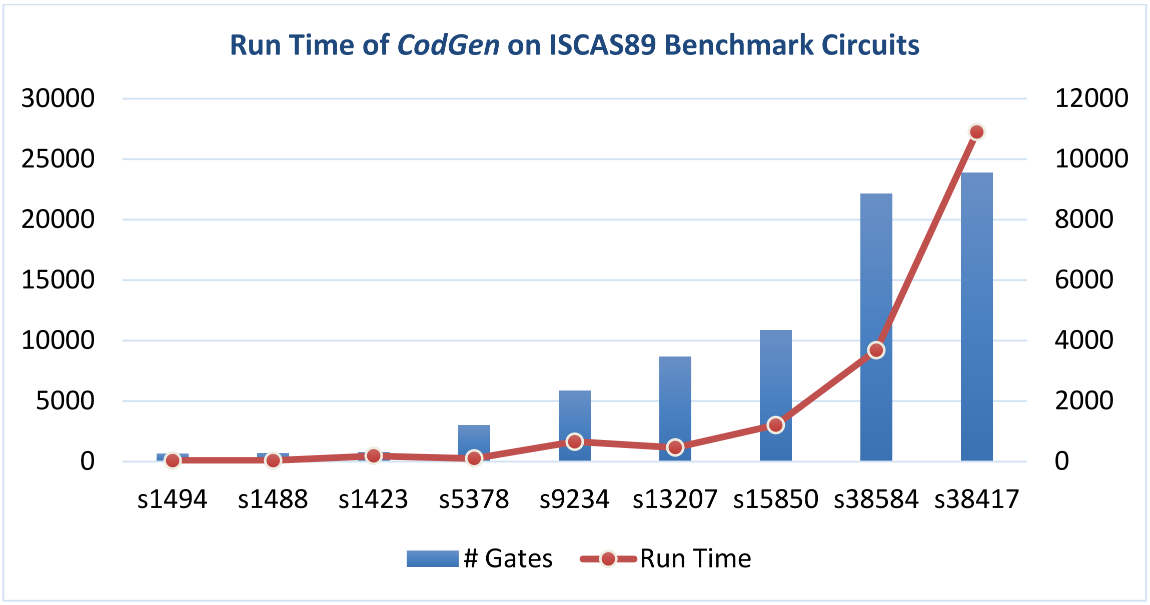


Figure 25. Run Time for CodGen on ISCAS89 Benchmark Circuits

CodGen has been run on the ISCAS89 benchmark circuits with *K* values from 1 to 5 and CYC values of 1, 2 and 3. A total of 15 *CodGen* runs with these combinations of *K* and CYC values have been run on each of the benchmark circuits (see Appendix I). The trend of the total run time and the number of paths is similar across circuits.

The total number of generated paths increases with the total number of gates in the circuit. But it should be noted that structure of the circuit can influence the path generation and may lead of lower number of paths for a circuit with relatively higher number of gates in the design. Logical connection in a relatively bigger circuit can lead to higher number of direct implication failures that may eventually lower the total number of generated paths. All the data points in Table 2 were generated assuming all sequential elements in the design to be SFF type. *Codgen* has been validated by modifying a selected group of sequential elements to non-scan cells, and the reduction in the number of paths verified.

Table 3 and Table 4 capture the data for configurations of *K*=1 and CYC=2 and 3 respectively.

Table 3. CodGen Run for Configuration: K=1, CYC=2

CodGen Run Configuration: K=1 CYC=2					
Benchmark Circuit	# Gates	# FF	# Paths	Longest Path Length	Run Time (s)
s1494	661	6	44	34	41
s1488	667	6	52	33	45
s1423	748	74	26	42	217
s5378	2993	179	225	30	138
s9234	5844	211	123	64	653
s13207	8651	639	67	66	731
s15850	10833	534	139	75	1764
s38584	22142	2426	544	84	4270
s38417	23843	2636	603	68	11812

Table 4. CodGen Run for Configuration: K=1, CYC=3

CodGen Run Configuration: K=1 CYC=3					
Benchmark Circuit	# Gates	# FF	# Paths	Longest Path Length	Run Time (s)
s1494	661	6	36	38	47
s1488	667	6	36	39	50
s1423	748	74	21	44	293
s5378	2993	179	201	35	146
s9234	5844	211	34	68	1088
s13207	8651	639	26	75	763
s15850	10833	534	61	100	1923
s38584	22142	2426	346	107	4382
s38417	23843	2636	186	91	13788

The graph of run time vs. total number of gates is drawn in the Figure 26. As can be seen, there is little difference in the run time for *CYC*=2 vs. *CYC*=3.

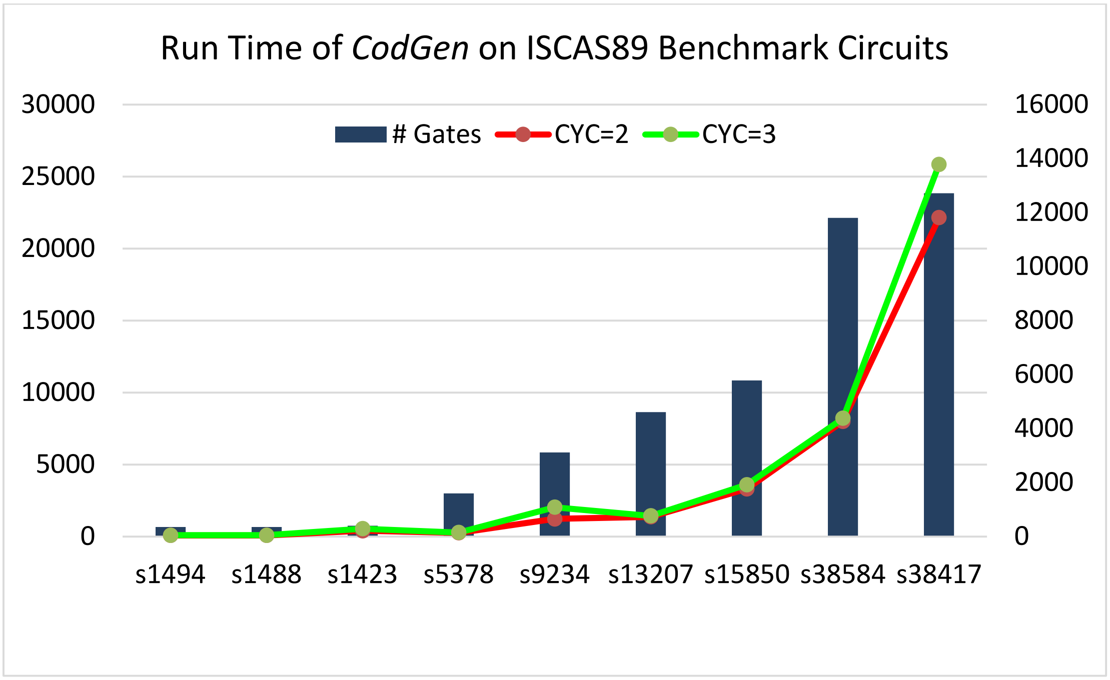


Figure 26. Run Time of CodGen on ISCAS89 Benchmark Circuits

Figure 27 captures the longest testable path length generated with different CYC values.

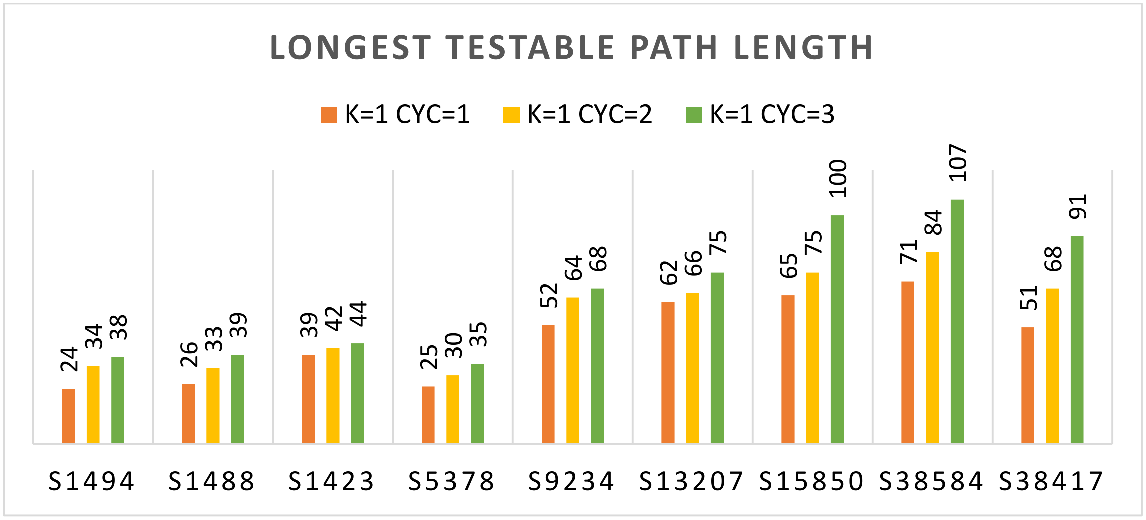
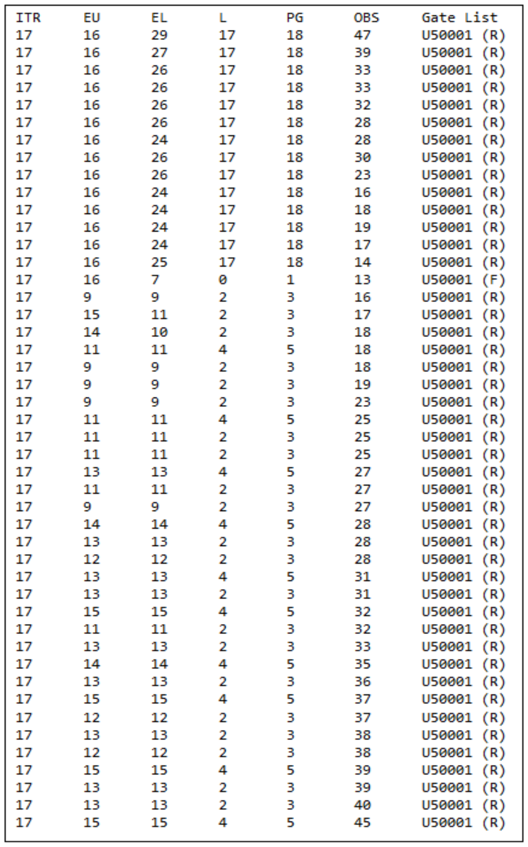


Figure 27. Longest Testable Path Length over Multiple Coda Cycles

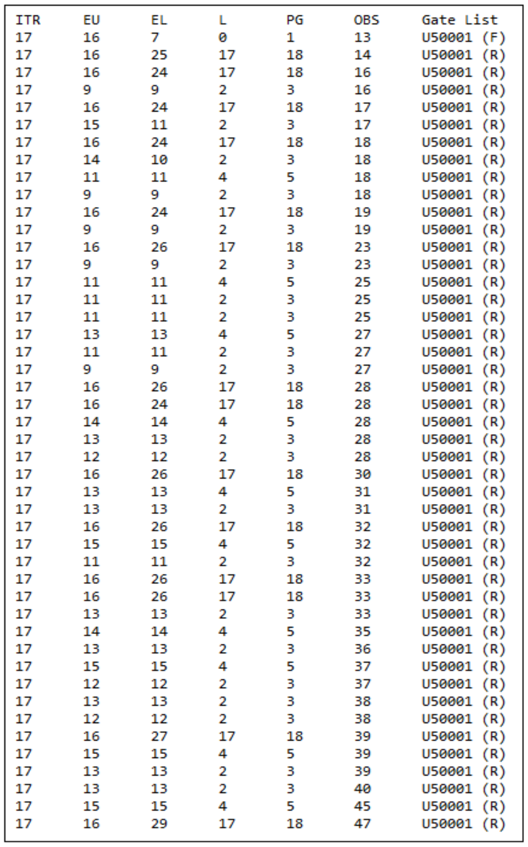
The longest testable path length increases over multiple code cycles for all the benchmark circuits. Circuit structure determines how long a path would be. Hence there is no direct relation between the longest paths across different circuits. The same behavior has been observed with higher *K* values as well.

5.2 Path Pool Reordering in *Coda* Cycle

Figure 28 captures the partial path pool ordering for one of the iterations during path generation.



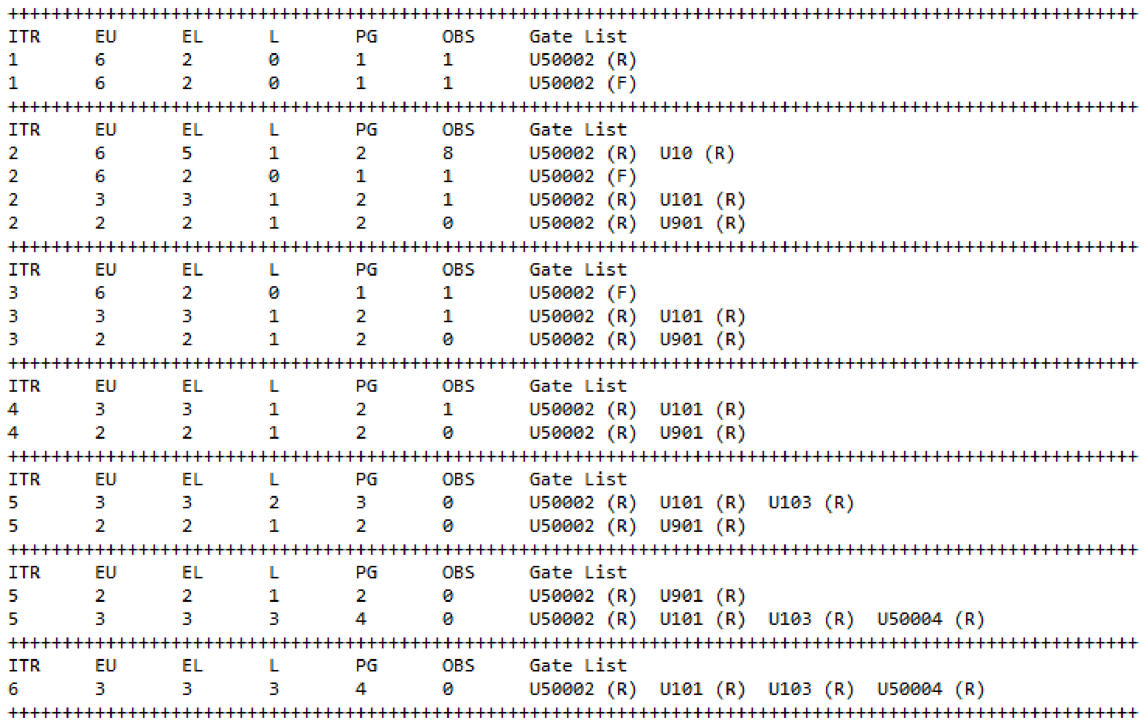
ITR	EU	EL	L	PG	OBS	Gate List
17	16	29	17	18	47	U50001 (R)
17	16	27	17	18	39	U50001 (R)
17	16	26	17	18	33	U50001 (R)
17	16	26	17	18	33	U50001 (R)
17	16	26	17	18	32	U50001 (R)
17	16	26	17	18	28	U50001 (R)
17	16	24	17	18	28	U50001 (R)
17	16	26	17	18	30	U50001 (R)
17	16	26	17	18	23	U50001 (R)
17	16	24	17	18	16	U50001 (R)
17	16	24	17	18	18	U50001 (R)
17	16	24	17	18	19	U50001 (R)
17	16	24	17	18	17	U50001 (R)
17	16	25	17	18	14	U50001 (R)
17	16	7	0	1	13	U50001 (F)
17	9	9	2	3	16	U50001 (R)
17	15	11	2	3	17	U50001 (R)
17	14	10	2	3	18	U50001 (R)
17	11	11	4	5	18	U50001 (R)
17	9	9	2	3	18	U50001 (R)
17	9	9	2	3	19	U50001 (R)
17	9	9	2	3	23	U50001 (R)
17	11	11	4	5	25	U50001 (R)
17	11	11	2	3	25	U50001 (R)
17	11	11	2	3	25	U50001 (R)
17	13	13	4	5	27	U50001 (R)
17	11	11	2	3	27	U50001 (R)
17	9	9	2	3	27	U50001 (R)
17	14	14	4	5	28	U50001 (R)
17	13	13	2	3	28	U50001 (R)
17	12	12	2	3	28	U50001 (R)
17	13	13	4	5	31	U50001 (R)
17	13	13	2	3	31	U50001 (R)
17	15	15	4	5	32	U50001 (R)
17	11	11	2	3	32	U50001 (R)
17	13	13	2	3	33	U50001 (R)
17	14	14	4	5	35	U50001 (R)
17	13	13	2	3	36	U50001 (R)
17	15	15	4	5	37	U50001 (R)
17	12	12	2	3	37	U50001 (R)
17	13	13	2	3	38	U50001 (R)
17	12	12	2	3	38	U50001 (R)
17	15	15	4	5	39	U50001 (R)
17	13	13	2	3	39	U50001 (R)
17	13	13	2	3	40	U50001 (R)
17	15	15	4	5	45	U50001 (R)



ITR	EU	EL	L	PG	OBS	Gate List
17	16	7	0	1	13	U50001 (F)
17	16	25	17	18	14	U50001 (R)
17	16	24	17	18	16	U50001 (R)
17	9	9	2	3	16	U50001 (R)
17	16	24	17	18	17	U50001 (R)
17	15	11	2	3	17	U50001 (R)
17	16	24	17	18	18	U50001 (R)
17	14	10	2	3	18	U50001 (R)
17	11	11	4	5	18	U50001 (R)
17	9	9	2	3	18	U50001 (R)
17	16	24	17	18	19	U50001 (R)
17	9	9	2	3	19	U50001 (R)
17	16	26	17	18	23	U50001 (R)
17	9	9	2	3	23	U50001 (R)
17	11	11	4	5	25	U50001 (R)
17	11	11	2	3	25	U50001 (R)
17	11	11	2	3	25	U50001 (R)
17	13	13	4	5	27	U50001 (R)
17	11	11	2	3	27	U50001 (R)
17	9	9	2	3	27	U50001 (R)
17	16	26	17	18	28	U50001 (R)
17	16	24	17	18	28	U50001 (R)
17	14	14	4	5	28	U50001 (R)
17	13	13	2	3	28	U50001 (R)
17	12	12	2	3	28	U50001 (R)
17	16	26	17	18	30	U50001 (R)
17	13	13	4	5	31	U50001 (R)
17	13	13	2	3	31	U50001 (R)
17	16	26	17	18	32	U50001 (R)
17	15	15	4	5	32	U50001 (R)
17	11	11	2	3	32	U50001 (R)
17	16	26	17	18	33	U50001 (R)
17	16	26	17	18	33	U50001 (R)
17	13	13	2	3	33	U50001 (R)
17	14	14	4	5	35	U50001 (R)
17	13	13	2	3	36	U50001 (R)
17	15	15	4	5	37	U50001 (R)
17	12	12	2	3	37	U50001 (R)
17	13	13	2	3	38	U50001 (R)
17	12	12	2	3	38	U50001 (R)
17	16	27	17	18	39	U50001 (R)
17	15	15	4	5	39	U50001 (R)
17	13	13	2	3	39	U50001 (R)
17	13	13	2	3	40	U50001 (R)
17	15	15	4	5	45	U50001 (R)
17	16	29	17	18	47	U50001 (R)

Figure 28. Path Pool Reordering in Coda Cycle

Each line in these tables represent a partial path. Figure 29 illustrates the path structure in the partial path pool. Only one gate is shown in Figure 28 due to space restriction.



```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
ITR     EU      EL      L       PG      OBS     Gate List
1       6       2       0       1       1       U50002 (R)
1       6       2       0       1       1       U50002 (F)
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
ITR     EU      EL      L       PG      OBS     Gate List
2       6       5       1       2       8       U50002 (R)  U10 (R)
2       6       2       0       1       1       U50002 (F)
2       3       3       1       2       1       U50002 (R)  U101 (R)
2       2       2       1       2       0       U50002 (R)  U901 (R)
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
ITR     EU      EL      L       PG      OBS     Gate List
3       6       2       0       1       1       U50002 (F)
3       3       3       1       2       1       U50002 (R)  U101 (R)
3       2       2       1       2       0       U50002 (R)  U901 (R)
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
ITR     EU      EL      L       PG      OBS     Gate List
4       3       3       1       2       1       U50002 (R)  U101 (R)
4       2       2       1       2       0       U50002 (R)  U901 (R)
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
ITR     EU      EL      L       PG      OBS     Gate List
5       3       3       2       3       0       U50002 (R)  U101 (R)  U103 (R)
5       2       2       1       2       0       U50002 (R)  U901 (R)
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
ITR     EU      EL      L       PG      OBS     Gate List
5       2       2       1       2       0       U50002 (R)  U901 (R)
5       3       3       3       4       0       U50002 (R)  U101 (R)  U103 (R)  U50004 (R)
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
ITR     EU      EL      L       PG      OBS     Gate List
6       3       3       3       4       0       U50002 (R)  U101 (R)  U103 (R)  U50004 (R)
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Figure 29. Partial Path Pool Expansion in *CodGen*

It can be seen that the length of the partial path increases upon successful addition of a new gate (success in direct implication). When a path cannot be grown any further through a branch, path growth algorithm returns to the last successful node from which the path split due to more than one fan-out. The left table in Figure 28 shows the path pool in one of the at-speed cycles. It be seen that for iteration 17 of path growth from the SFF U50001, the path pool is sorted according to the *Esperance* values. The right side table shows the same iteration with observability driven ordering of the path pool.

5.3 *CodGen* Run Time Profile

The run time code profiling data is illustrated in Figure 30 for *s38584* benchmark with a run time configuration of *K*=1 and *CYC*=1 and robust sensitization. Two major components of the total run time are the *Aiming Path Generation* and the *Path Expansion* subroutine.

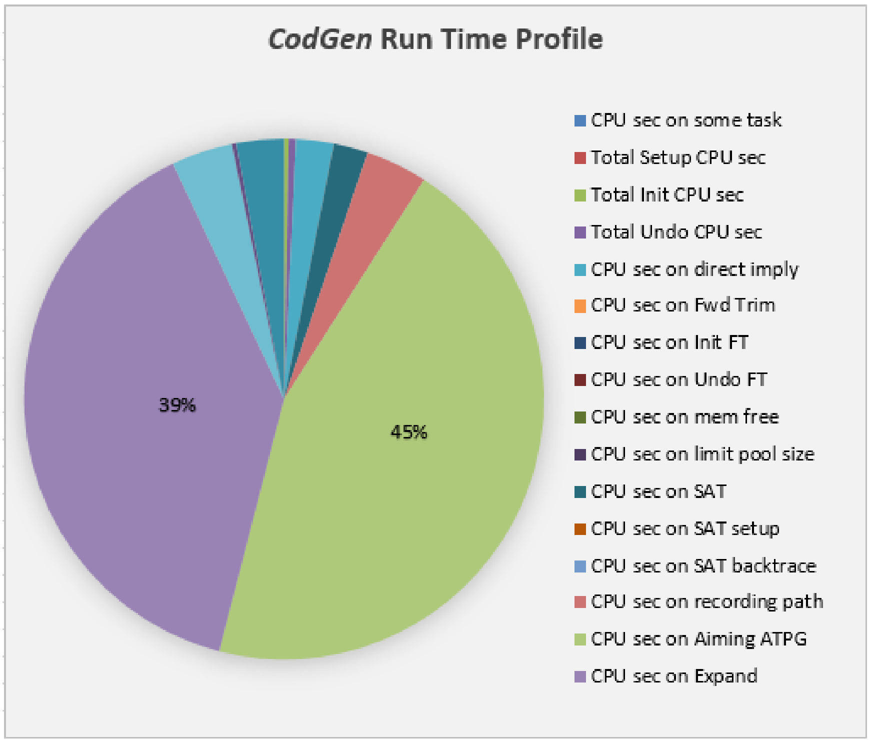


Figure 30. CodGen Run Time Profiling for *s38584*

The third most time consuming function is the SAT based justification and compaction of a complete path. For a benchmark circuit such as *s38584,* a considerable amount of time goes behind reporting the generated path as well. For a relatively smaller circuit with many fewer generated paths, the reporting function takes a much lower share of the total run time.

The distribution of individual functions on the total run time plot is uniform across *K* and *CYC* values. Figure 31 captures the run time distribution for *s1494* and *s38584* for two different configurations of *K* and *CYC* values.

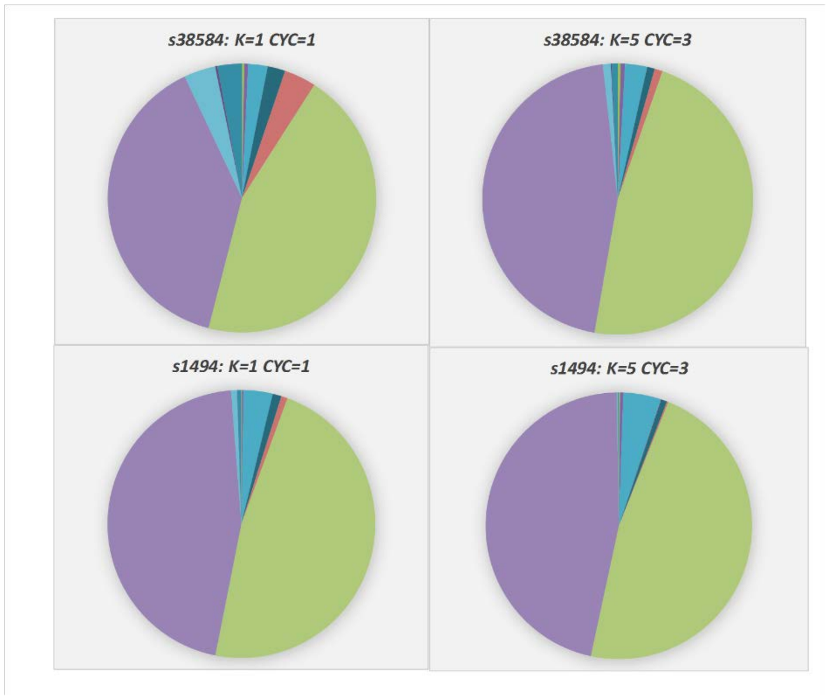


Figure 31. *CodGen* Run Time Profiling across Multiple Run Configurations

The circuit size and run configuration used to plot the graph in Figure 31 encompass the total experimental setup in terms of total run time and total number of generated paths. Hence, this plot gives a good idea about the *CodGen* profiling based on its run time configuration across all feasible possibilities.

For the same benchmark circuit, the total time spent behind *Aiming Path Generation* and *Path Expansion* is higher for larger *K* and *CYC* values. This is an expected behavior since *CodGen* deals with a larger search space with bigger *K* and *CYC* values.

6. CONCLUSIONS AND FUTURE WORK

The work presented in this thesis enables observability driven path generation for *coda* cycles. The *CodGen* is enhanced with path growth metric driven distinction between the at-speed and *coda* cycles. The implementation ensures that no false path is generated in the presence of non-scan FF at the capture point.

Inclusion of SFF in a circuit incurs area growth. Area critical SOC solutions may have an upper bound for adding scan chains in the design. Since *Esperance* based path growth does not differentiate between a SFF and a regular FF, false paths would be generated in a scan-limited circuit. As a result, test coverage would be limited. On the other hand, observability driven path generation in *coda* cycle can regain the loss in coverage. The extent of coverage gain by this process can be limited by structure of the circuit. Low fan-in of FF would limit path growth through other branches which may terminate at a SFF.

As a part of the future work, the *coda* cycle path growth can be characterized with larger circuits. Experiments with higher values of *K* and *CYC* can be performed to confirm the trend in run time and number of paths generated as seen with the current experimental results.

Test coverage analysis can be carried out to understand the quantitative impact of moving from *Esperance* based path growth to the observability driven path growth in the *coda* cycles.

Transitions launched in the at-speed cycle can be lost at memory boundary. The memory array can be modeled as a non-scan element. Hence the observability driven path generation can be applied to memory shadow logic coverage.

Preamble cycles have not been considered in the path generation experiments in the thesis. It would be a good exercise to include these pseudo functional cycles to cover the entire clocking scheme in one single *CodGen* setup. The logic state at the PI nodes have been considered to be constant. A time varying input vector can be used for further experiments.

All the experiments assume robust sensitization. The results can be generated for non-robust and long transition path sensitization as well. The primary objective of the work was to establish the observability metric for *coda* cycles. Hence, robust test vector scenario validates the concept. Non-robust test generation would differ primarily in the number of paths that can be generated over multiple *coda* cycles. However, the effect of using observability over *Esperance* would be same as in the case of robust test.

New path growth heuristics can be explored to improve the run time of *CodGen.* Direct implication and partial path expansion are two of the primary functions that consume most of the run time. A different approach for storing the partial path in a more efficient data structure can be explored as one of the ways to improve the run time.

REFERENCES

[1] Tehranipoor, Mohammad, Peng, Ke, Chakrabarty, Krishnendu, "Test and Diagnosis for Small-Delay Defects", Springer Science+Business Media, LLC 2011.

[2] Pomeranz, I.; Reddy, S.M., "Transition Path Delay Faults: A New Path Delay Fault Model for Small and Large Delay Defects," in Very Large Scale Integration (VLSI) Systems, IEEE Transactions on , vol.16, no.1, pp.98-107, Jan. 2008

[3] G. L. Smith, "Model for Delay Faults Based Upon Paths," IEEE International Test Conference, Oct. 1985, pp. 342-349.

[4] Laung-Terng Wang, Charles Stroud, Nur Touba "System-on-Chip Test Architectures, 1st Edition Nanometer Design for Testability", Morgan Kaufmann 2010.

[5] M. L. Bushnell, V. D. Agrawal, "Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits," Springer 2000.

[6] W. N. Li, S. M. Reddy, S. K. Sahni, "On Path Selection in Combinational Logic Circuits," IEEE Trans. On Computer-Aided Design, vol. 8, no. 1, Jan. 1989, pp.56-63.

[7] A. K. Majhi, V. D. Agrawal, J. Jacob, L. M. Patnaik, "Line Coverage of Path Delay Faults," IEEE Trans. on VLSI Systems, vol. 8, no. 5, Oct. 2000, pp. 610-613.

[8] A. Murakami , S. Kajihara, T. Sasao, I. Pomeranz, S.M. Reddy, "Selection of Potentially Testable Path Delay Faults for Test Generation," IEEE International Test Conference, 2000, pp. 376-384.

[9] Y. Shao, S.M. Reddy, I. Pomeranz, S. Kajihara, "On Selecting Testable Paths in Scan Designs," IEEE European Test Workshop, 2002, pp. 53-58.

[9] K. Fuchs, F. Fink, M. H. Schulz, "DYNAMITE: An Efficient Automatic Test Pattern Generation System for Path Delay Faults," IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, 1991, vol.10, no.10, pp.1323-1335.

[10] M. Sharma, J. H. Patel, "Finding a Small Set of Longest Testable Paths that Cover every Gate," IEEE International Test Conference, 2002, pp. 974-982.

[11] W. Qiu, D. M. H. Walker, "An Efficient Algorithm for Finding the K Longest Testable Paths Through Each Gate in a Combinational Circuit", IEEE International Test Conference, Sept. 2003, pp. 592-601.

[12] Drego, N.; Chandrakasan, A.; Boning, D., "An all-digital, highly scalable architecture for measurement of spatial variation in digital circuits," in Solid-State Circuits Conference, 2008. A-SSCC '08. IEEE Asian , vol., no., pp.393-396, 3-5 Nov. 2008

[13] J. Liou, L.-C. Wang, K.-T. Cheng, "On Theoretical and Practical Considerations of Path Selection for Delay Fault Testing," Proc. IEEE/ACM International Conference on Computer Aided Design, 2002, pp. 94-100.

[14] C. Lin, S. Reddy, "On Delay Fault Testing in Logic Circuits," IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, vol.6, no.5, Sept. 1987, pp. 694-703.

[15] P. McGeer, R. Brayton, "Efficient Algorithms for Computing the Longest Viable Path in a Combinational Network," Proc. ACM/IEEE Design Automation Conference, June 1989, pp. 561-567.

[16] J. Benkoski, E. Meersch, L. Claesen, H. Man, "Timing Verification using Statically Sensitizable Paths," IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, , vol.9, no.10, Oct. 1990, pp.1073-1084.

[17] H. Chang, J. Abraham, "VIPER: An Efficient Vigorously Sensitizable Path Extractor," Proc. ACM/IEEE Design Automation Conference, June 1993, pp.112-117.

[18] J. Liou, A. Krstic, L.-C. Wang, K.-T. Cheng, "False-path-aware Statistical Timing Analysis and Efficient Path Selection for Delay Testing and Timing Validation," Proc. ACM/IEEE Design Automation Conference, 2002, pp. 566-569.

[19] Furukawa, H.; Wen, X.; Miyase, K.; Yamato, Y.; Kajihara, S.; Girard, P.; Wang, L.-T.; Tehranipoor, M., "CTX: A Clock-Gating-Based Test Relaxation and X-Filling Scheme for Reducing Yield Loss Risk in At-Speed Scan Testing," in Asian Test Symposium, 2008. ATS '08. 17th , vol., no., pp.397-402, 24-27 Nov. 2008

[20] Bonhomme, Y.; Girard, P.; Guiller, L.; Landrault, C.; Pravossoudovitch, S., "A gated clock scheme for low power scan testing of logic ICs or embedded cores," in Test Symposium, 2001. Proceedings.10thAsian,vol.,no.,pp.253-258,2001.

[21] W. Qiu, J. Wang, D. M. H. Walker, D. Reddy, X. Lu, Z. Li, W. Shi and H. Balachandran, “K Longest Paths Per Gate (KLPG) Test Generation for Scan- Based Sequential Circuits,” IEEE International Test Conference, Oct. 2004, pp. 223-231.

[22] P. Pant, J. Zelman, “Understanding Power Supply Droop During At-Speed Scan Testing,” IEEE VLSI Test Symposium, May 2009, pp.227-232.

[23] Chakraborty, S.; Walker, D.M.H., "At-Speed Path Delay Test," in Test Workshop (NATW), 2015 IEEE 24th North Atlantic , vol., no., pp.39-42, 11-13 May 2015.

[24] Yung-Chieh Lin; Feng Lu; Kai Yang; Kwang-Ting Cheng, "Constraint extraction for pseudo-functional scan-based delay testing," in Design Automation Conference, 2005. Proceedings of the ASP-DAC 2005. Asia and South Pacific , vol.1, no., pp.166-171 Vol. 1, 18-21 Jan. 2005

[25] Zhaohui Fu; Yinlei Yu; Malik, S., "Considering circuit observability don't cares in CNF satisfiability," in Design, Automation and Test in Europe, 2005. Proceedings , vol., no., pp.1108-1113 Vol. 2, 7-11 March 2005

[26] K. Bian, D. M. H. Walker, S. Khatri, S. Lahiri, "Mixed Structural-Functional Path Delay Test Generation and Compaction," IEEE International Symposium Defect and Fault Tolerance in VLSI and Nanotechnology Systems, Oct. 2013, pp. 7-12.

[27] N. Eén, N. Sörensson, "The MiniSat Page, Introduction". Retrieved from minisat.se on March 2015.

[28] Gonciari, P.T.; Al-Hashimi, B.M.; Nicolici, N., "Improving compression ratio, area overhead, and test application time for system-on-a-chip test data compression/decompression," in Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings, vol., no., pp.604-611, 2002

[29] T. Larrabee, "Test Pattern Generation Using Boolean Satisfiability", *IEEE Trans. Computer*-Aided Design, vol. 11, no. 1, pp. 4-15, Jan. 1992.

[30] Menon, P.; Levendel, Y.; Abramovici, M., "SCRIPT: a critical path tracing algorithm for synchronous sequential circuits," in Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on , vol.10, no.6, pp.738-747, Jun 1991

[31] Nisar Ahmed, Mohammad Tehranipoor, "Improving Transition Delay Fault Coverage Using Hybrid Scan-Based Technique," 2012 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), pp. 187-198, 20th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'05), 2005

APPENDIX – I

Results with Various *K* and CYC Values for each Benchmark Circuit

Table 5 Run Time and Generated Paths for *s1494*

	Run Time (h:m:s)	*CYC*=1	*CYC*=2	*CYC*=3	# Paths	*CYC*=1	*CYC*=2	*CYC*=3
S1494	K=1	0:00:29	0:00:41	0:00:47	K=1	55	44	36
	K=2	0:00:47	0:01:01	0:01:12	K=2	124	86	49
	K=3	0:01:08	0:01:07	0:01:21	K=3	153	108	58
	K=4	0:01:14	0:01:16	0:01:32	K=4	180	113	62
	K=5	0:01:39	0:01:47	0:01:50	K=5	214	129	68

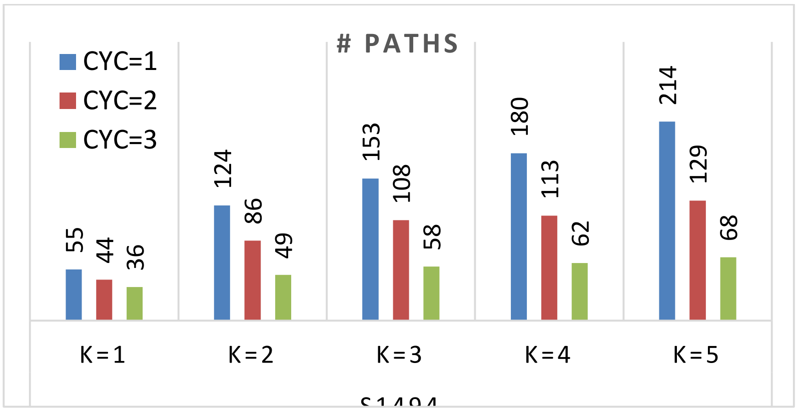


Figure 32. # Paths generated for *s1494*

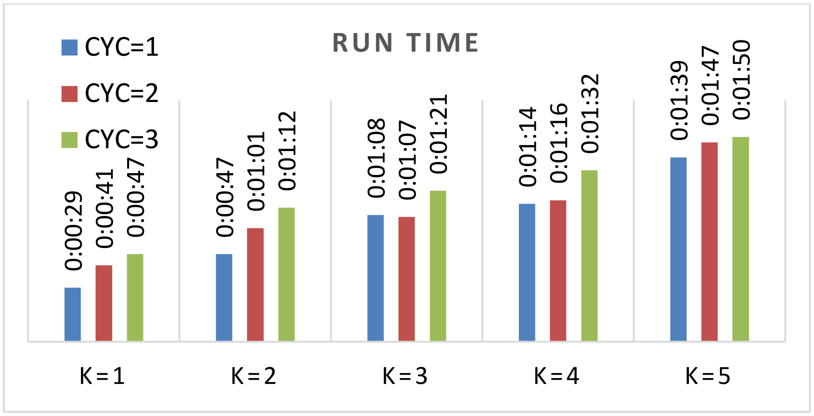


Figure 33. Run Time for *s1494*

Table 6 Run Time and Generated Paths for s1423

	Run Time (h:m:s)	*CYC*=1	*CYC*=2	*CYC*=3	# Paths	*CYC*=1	*CYC*=2	*CYC*=3
S1423	K=1	0:03:05	0:03:37	0:04:53	K=1	73	26	19
	K=2	0:03:22	0:03:52	0:05:28	K=2	156	38	21
	K=3	0:03:53	0:05:51	0:05:38	K=3	209	43	21
	K=4	0:04:51	0:05:42	0:06:33	K=4	228	43	21
	K=5	0:05:50	0:05:38	0:05:36	K=5	233	43	21

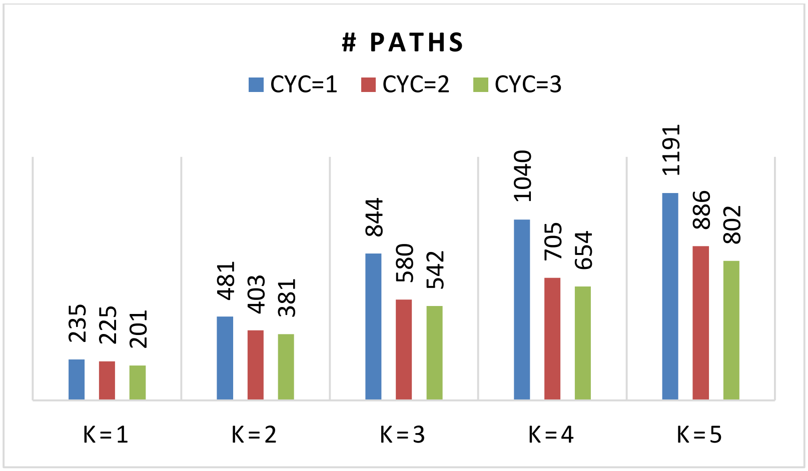


Figure 34. # Paths for *s1423*

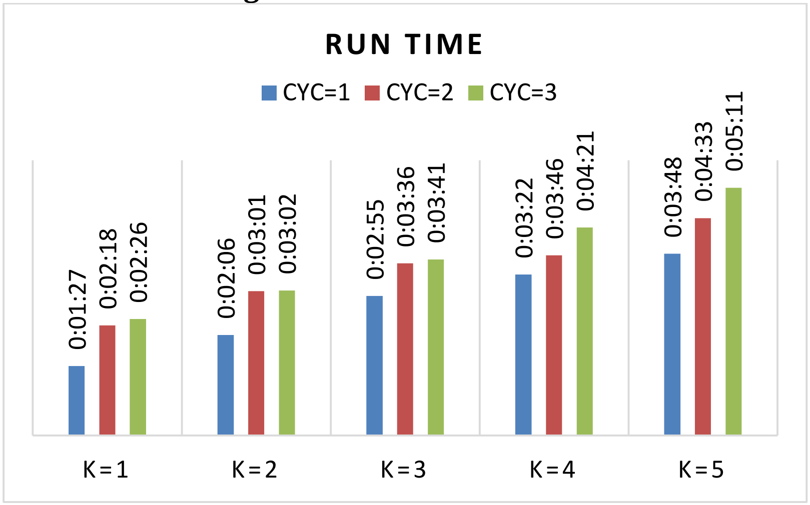


Figure 35. Run Time for *s1423*

Table 7 Run Time and Generated Paths for s5378

	Run Time (h:m:s)	*CYC*=1	*CYC*=2	*CYC*=3	# Paths	*CYC*=1	*CYC*=2	*CYC*=3
S5378	K=1	0:01:27	0:02:18	0:02:26	K=1	235	225	201
	K=2	0:02:06	0:03:01	0:03:02	K=2	481	403	381
	K=3	0:02:55	0:03:36	0:03:41	K=3	844	580	542
	K=4	0:03:22	0:03:46	0:04:21	K=4	1040	705	654
	K=5	0:03:48	0:04:33	0:05:11	K=5	1191	886	802

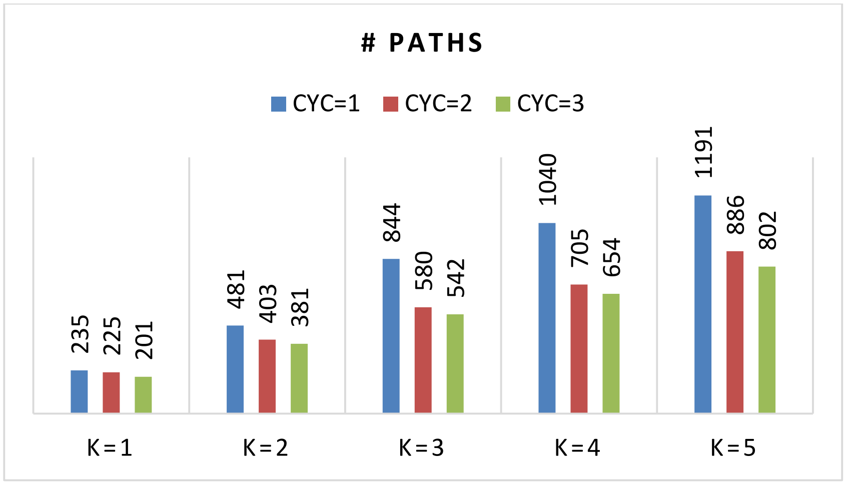


Figure 36. # Paths for *s5378*

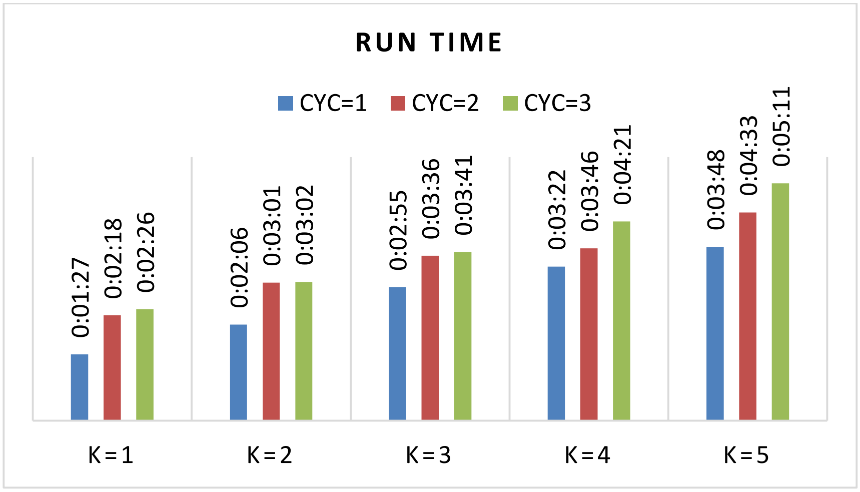


Figure 37. Run Time for *s5378*

Table 8 Run Time and Generated Paths for *s9234*

	Run Time (h:m:s)	*CYC*=1	*CYC*=2	*CYC*=3	# Paths	*CYC*=1	*CYC*=2	*CYC*=3
S9234	K=1	0:10:52	0:10:53	0:18:08	K=1	143	123	34
	K=2	0:12:49	0:13:04	0:18:07	K=2	295	203	58
	K=3	0:14:03	0:13:36	0:19:42	K=3	419	299	70
	K=4	0:14:37	0:14:44	0:20:04	K=4	513	355	71
	K=5	0:15:13	0:14:36	0:20:07	K=5	609	445	71

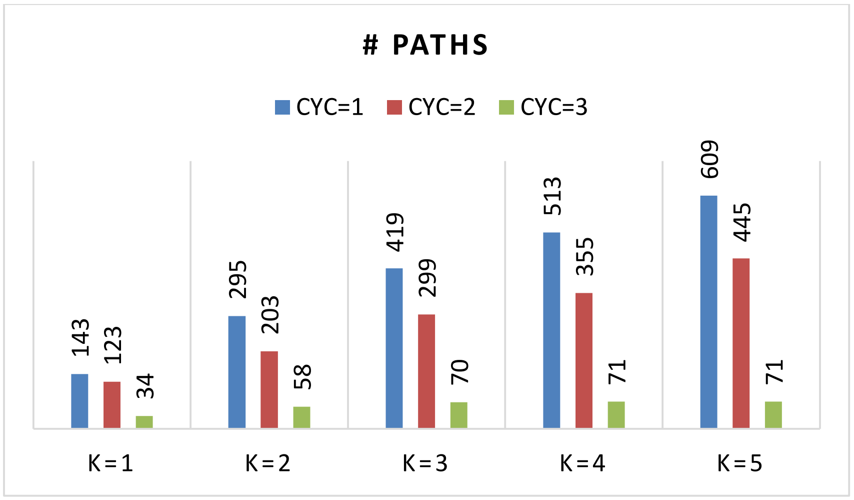


Figure 38. # Paths for *s9234*

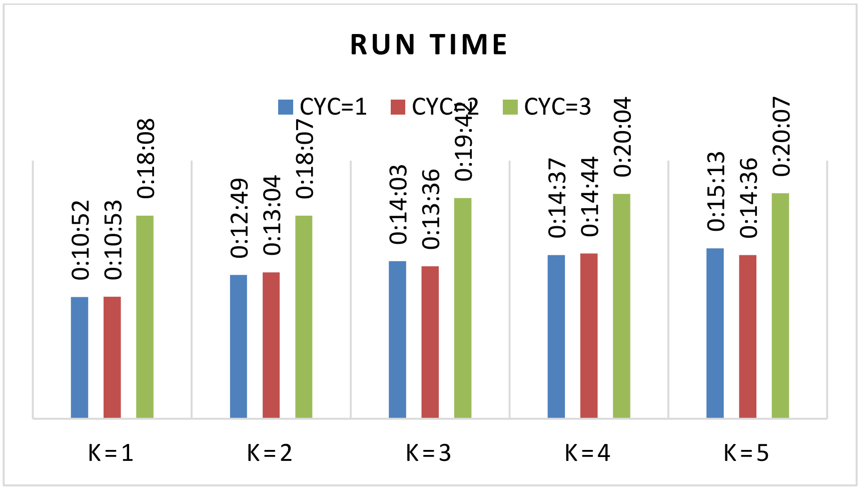


Figure 39. Run Time for *s9234*

Table 9 Run Time and Generated Paths for *s13207*

	Run Time (h:m:s)	*CYC*=1	*CYC*=2	*CYC*=3	# Paths	*CYC*=1	*CYC*=2	*CYC*=3
S13207	K=1	0:07:38	0:12:11	0:12:43	K=1	224	67	26
	K=2	0:10:34	0:13:18	0:12:52	K=2	426	85	30
	K=3	0:10:29	0:13:15	0:12:53	K=3	520	88	30
	K=4	0:11:41	0:13:31	0:13:00	K=4	553	89	30
	K=5	0:13:16	0:13:23	0:13:00	K=5	601	89	30

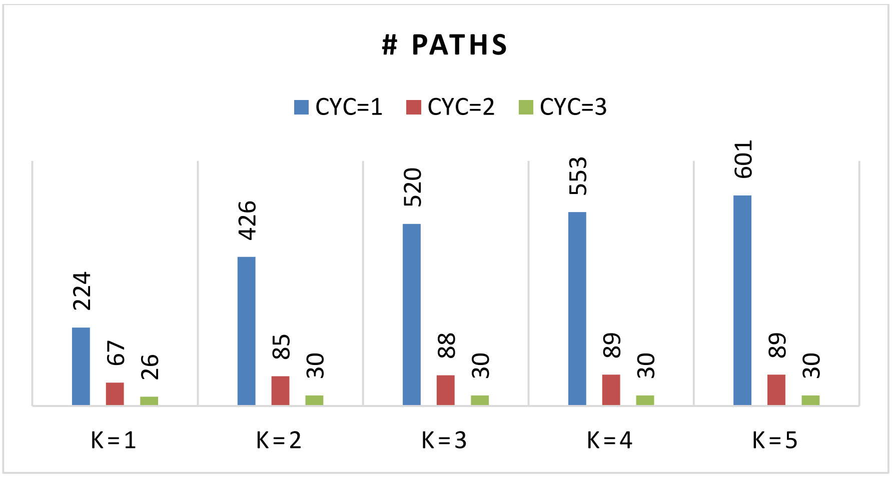


Figure 40. # Paths for *s13207*

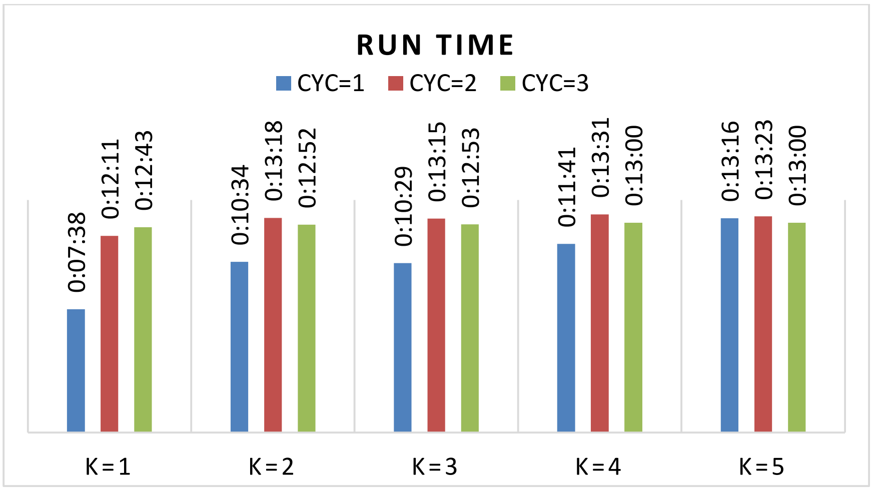


Figure 41. Run Time for *s13207*

Table 10 Run Time and Generated Paths for *s15850*

	Run Time (h:m:s)	*CYC*=1	*CYC*=2	*CYC*=3	# Paths	*CYC*=1	*CYC*=2	*CYC*=3
S15850	K=1	0:20:06	0:29:24	0:32:03	K=1	442	139	61
	K=2	0:26:13	0:31:23	0:26:31	K=2	584	160	72
	K=3	0:29:00	0:26:06	0:26:28	K=3	617	160	80
	K=4	0:26:52	0:28:20	0:27:51	K=4	617	160	80
	K=5	0:27:24	0:29:29	0:27:16	K=5	617	160	80

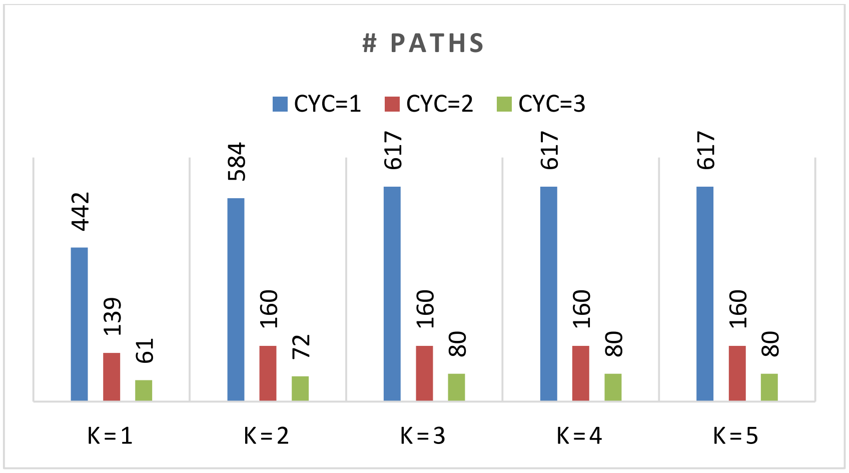


Figure 42. # Paths for *s15850*

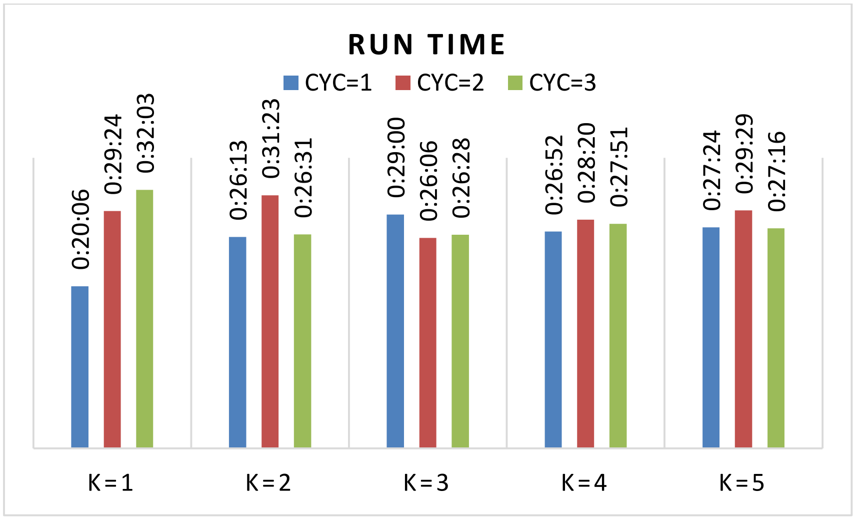


Figure 43. Run Time for *s15850*

Table 11 Run Time and Generated Paths for *s38584*

	Run Time (h:m:s)	*CYC*=1	*CYC*=2	*CYC*=3	# Paths	*CYC*=1	*CYC*=2	*CYC*=3
S38584	K=1	1:02:18	1:11:10	1:13:02	K=1	1294	544	346
	K=2	1:17:36	1:14:54	1:14:34	K=2	1768	647	368
	K=3	1:19:23	1:13:28	1:13:21	K=3	1800	647	389
	K=4	1:22:05	1:15:33	1:13:34	K=4	1800	657	393
	K=5	1:23:52	1:17:17	1:15:04	K=5	1800	657	393

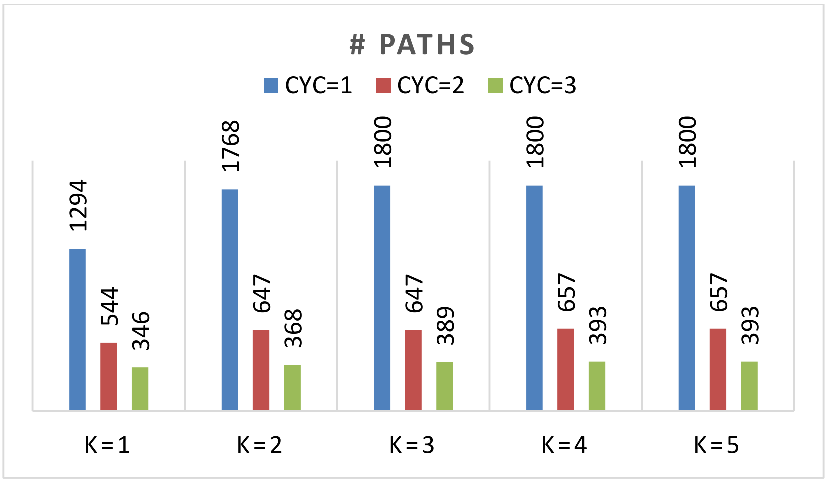


Figure 44. # Paths for *s38584*

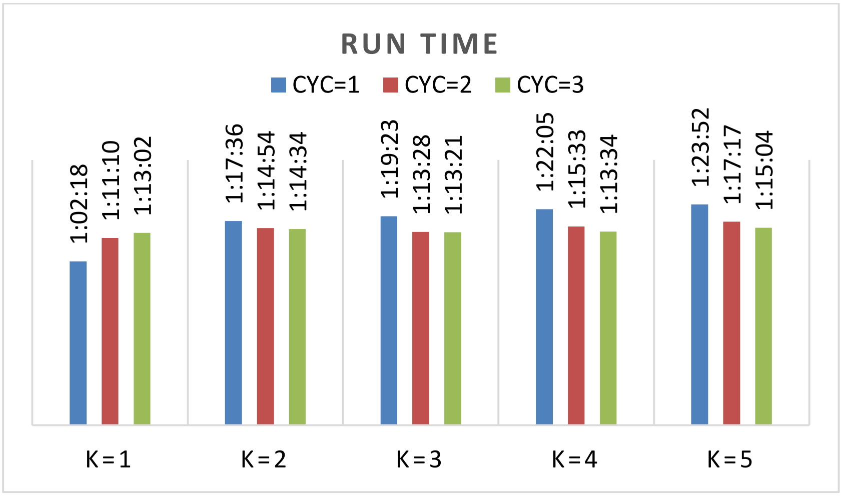


Figure 45. Run Time for *s38584*

Table 12 Run Time and Generated Paths for s38417

	Run Time (h:m:s)	*CYC*=1	*CYC*=2	*CYC*=3	# Paths	*CYC*=1	*CYC*=2	*CYC*=3
S38417	K=1	3:01:36	3:16:52	3:49:48	K=1	957	603	186
	K=2	3:40:38	4:04:00	3:20:53	K=2	1964	974	261
	K=3	4:00:28	4:03:01	3:57:21	K=3	2367	1209	340
	K=4	3:43:50	3:26:58	3:11:21	K=4	2542	1533	331
	K=5	4:44:40	4:28:06	3:56:21	K=5	2968	1910	336

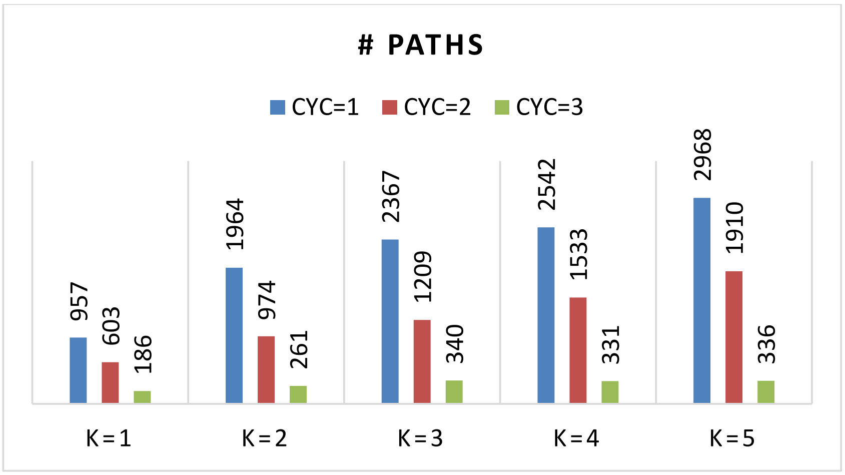


Figure 46. # Paths for *s38417*

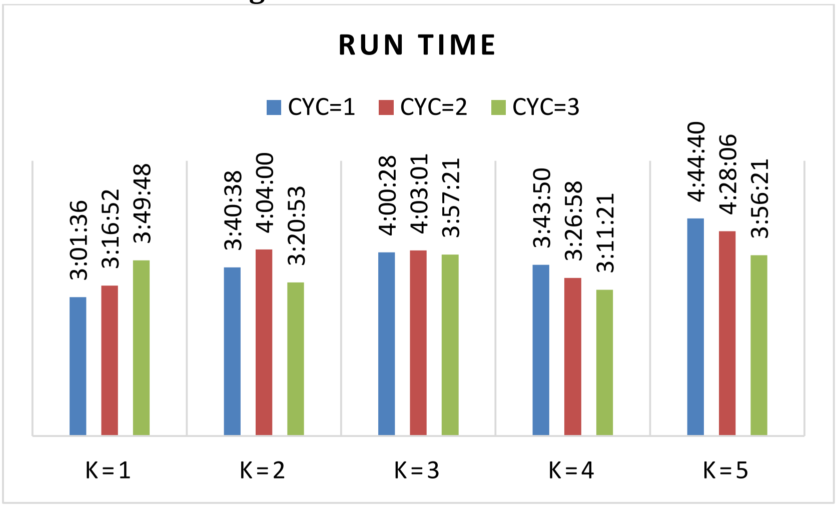


Figure 47. Run Time for *s38417*