OPTIMIZATION OF PSEUDO FUNCTIONAL PATH DELAY TEST THROUGH EMBEDDED MEMORIES

A Thesis

by

VISWANATH BOGA

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

Chair of Committee,	Duncan M. Walker
Committee Members,	Rabi N. Mahapatra
	Weiping Shi
Head of Department,	Dilma Da Silva

May 2016

Major Subject: Computer Science

ABSTRACT

Traditional automatic test pattern generation achieves high coverage of logic faults in integrated circuits. Automatic test of embedded memory arrays uses built-in self-test. Testing the memories and logic separately does not fully test the critical timing paths that go into or out of memories. Prior research has developed algorithms and software to test the longest paths into and out of embedded memories. However, in this prior work, the test generation time increased superlinearly with memory size. This is contrary to the intuition that the time should rise approximately linearly with memory size. This behavior limits the algorithm to circuits with relatively small memories. The focus of this research is to analyze the time complexity of the algorithm and propose changes to reduce the time required to test circuits with large memories.

We use our prior work on pseudo functional *K* longest path per gate test generation, and the benchmark circuits with embedded memories developed in the prior work. Since the cells within a memory array are not scan cells, a value that is captured in a memory cell must be moved to a scan cell using low-speed *coda* cycles. This approach will also support the test of any non-scan flip-flop or latch, in addition to embedded memory arrays. In addition to testing the critical timing paths, testing through memories eliminates the logic “shadows” around the memory where faults cannot be tested.

In this research our complexity analysis has identified the reason for the superlinear increase in test generation time with larger memories and verified this analysis with experimental results. We have also developed and implemented several heuristics to increase performance, with experimental results. This research also identifies the major algorithm changes required to further increase performance.

DEDICATION

To those who love me

ACKNOWLEDGEMENTS

I would like to thank my advisor and committee chair, Dr. Duncan M. (Hank) Walker for his advice and support throughout my M.S. studies at Texas A&M University. His insights in this particular research area, his technical guidance and spiritual support were invaluable to this work. This thesis would never have been completed without his advice and encouragement. I owe him lots of gratitude for making my research life enjoyable and rewarding. What I learned from him will benefit my future career.

I am grateful to my committee members, Dr. Rabi N. Mahapatra and Dr. Weiping Shi for their valuable suggestions and personal encouragement. I would also like to thank Weizhong Chen, and Yukun Gao, for their prior work, contributions and effective knowledge transfer regarding their developments in this area of research to me.

At the end, thanks for my mother, my father and my friends & family for what they have given to me.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

1.1. Delay Testing

Manufacturing process introduces defects in digital circuits. The long chain of steps involved in the fabrication of semiconductor chips may cause opens and shorts in the electrical lines which result in functional failure of the circuits. Traditional test methods [1][2][3] are used to detect these faults. However, the traditional methods will not detect some of the defects which only affect the operating speed of the circuit. It is necessary to detect these defects to ensure that the circuit is obeying the operating frequency timing constraints. These defects are small and could be detected with only delay tests. As the processes of manufacturing chips become more complex and the circuit operating frequency increases, there is more scope for the occurrence of these delay defects. There are two kinds of delay faults namely global delay faults and local delay faults. While global process parameter variations are responsible for the former, disturbances in the local process cause the later. Delay fault models [4][5] are developed in software to simulate Automatic Test Path Generation (ATPG)[6][7]. They also provide the fault coverage estimation [8]. Delay fault models represent various forms of delay defects that can occur in a circuit. Section 1.2 presents various delay fault models available in the literature. Our focus in this research would be on Small Delay Defaults (SDDs). Necessary design for delay testing and modifications required in the circuitry are presented in Section 1.3. Section 1.4 introduces traditional techniques used in testing circuits with memory. Section 1.5 explains the K Longest Path per Gate (KLPG) test approach. Section 1.6 presents the idea of Pseudo-Functional testing and why it is required. Section 1.7 details the structure of this thesis.

1.2. Delay Fault Models

The difference in the pre-silicon and post-silicon models of the circuit represents a defect. A fault is a functional representation of a defect. The changed behavior is the superficial observation of a fault. In presence of a defect, the expected outputs of the actual circuit are different from the expected outputs from the pre-silicon model. A few of the important delay fault models are presented below.

1.2.1. Transition Fault Model

The transition fault (TF) model [7] is the most popularly used model. In this model, the number of faults is linearly proportional to the number of gates. Every line in a circuit possibly can have either slow-to-rise (STR) or slow-to-fall (STF) transition fault. All lines connecting the gates are considered here. Hence, any line contributing to an input or an output of a gate in the circuit can have these faults. However, the additional delay caused by the fault is assumed to be large enough to delay the transition beyond the specified time from reaching the primary observable points. Any path to observable output, irrespective of the length can be used to observe this fault. Hence, the circuit timing need not be considered in the process of transition fault test generation.

Test generation tools build for stuck-at faults can be enhanced to generate the test vectors required for this test [2]. A vector pair {A, B} can be formed by pairing the test patterns for stuck-at-0 and stuck-at-1. The order of stuck-at-0 and stuck-at-1 test patterns in the vector pair can be changed to simulate STR or STF transition fault on a particular line. This is possible since every stuck-at fault is a transition fault with infinitely large delay. In the vector pair {A, B}, A initializes the circuit, B sensitizes the fault and propagates the effect to observable point.

This model has the shortcoming of amount of delay added by the fault not being considered. If the transition fault at a particular site needs to be tested, the shortest or the easiest path from the fault site to an observable primary output meets the requirement to simulate transition fault test. Hence, small delay defects cannot be appropriately tested using this fault model [9][10]. TF test can propagate a glitch from the fault site [11]. This affects the quality of the test.

1.2.2. Gate Delay Fault Model

Gate delay fault model [6][7][8][12] tries to sensitize the extra delay latched on to an input or output of a gate. The size of the extra delay is noted. A long path through the target fault site is used to propagate the transition and checked for the maximum extra delay added. The delay fault size should to be specified prior since the least detected delay fault size needs to be as close as possible to the minimum detectable fault sizes.

1.2.3. Line Delay Fault Model

Line delay fault model [13][14] is a variation to the previously mentioned model. the longest sensitizable path for every line in the circuit is used to test a rising or falling delay fault. The smallest delay defect can only be detected by targeting the longest path through a fault site.

1.2.4. Path Delay Fault Model

Path Delay fault model [5] is based on the accumulated delay on a path. This fault space is each and every possible path in the circuit. Intuitively larger paths can have larger delays. Small delay defects can be thus detected by testing the longest true paths in the circuit. However, the smallest delay fault can only be known by testing all the possible small paths as well. The number of paths in the circuit is an explosion in terms of number of gates and number of lines in the circuit. The path explosion needs to be considered before using this model on a circuit. For instance, ISCAS85 benchmark circuit c6288, a 16-bit multiplier, has close to 10^{20} paths [17]. Hence, testing all the paths is not pragmatic. The fault coverage is limited by the number of paths considered for testing. For the type of circuits where the number of paths is not exponential in the number of gates, this fault model can still be used.

1.2.5. Scan Based Delay Test

The cost of testing a digital logic circuit is represented by the Testability measure. The controllability and observability measures [15] calculated for each line of the circuit are used to analyze the testability. The testability of a circuit is higher if most of lines in the circuit can be controlled and if we have enough points to observe the outputs. Circuit needs to redesigned to achieve this. These techniques constitute Design-For-Test (DFT). One of the popular examples is scan design, shown in Figure 1. Scan cells are storage elements which are formed into chains connecting to various gates in the circuit to serve as pseudo primary inputs. They can also be used to observe the outputs of the circuit.

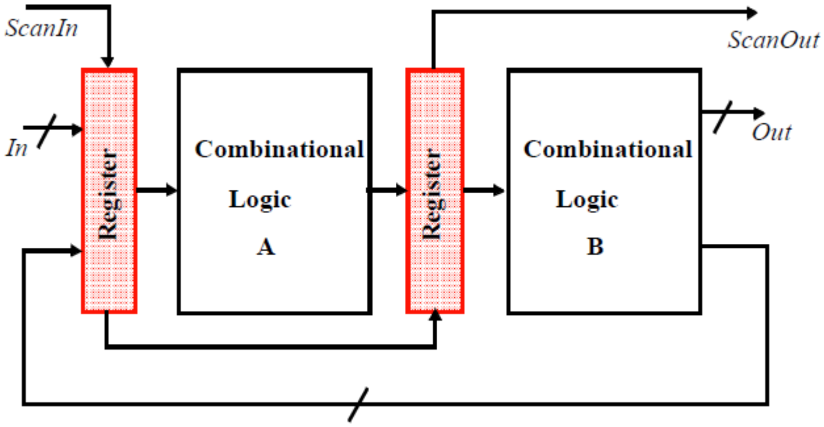


Figure 1. Structure of Scan Design [50]

The Circuit under test is set to scan mode, where the test vector is shifted using shift cycles (low power compared to at-speed). The scan chains can have multiple input points for filling the scan cells in the circuit. The circuit is placed in functional mode with at-speed clock cycles. The test results will be captured in scan cells and at primary outputs. The results are scanned out simultaneously as the next test vector is scanned in. The scan chains while providing control over internal lines of the circuit need to be properly fit into the circuit to achieve maximum fault coverage. The complexity of test pattern generation is reduced with scan design.

1.3. Scan Cell Types

1.3.1. A Muxed-D Scan

An edge-triggered muxed-D scan cell design is shown in Figure 2. It constitutes a 2x1 multiplexer and a regular D flip-flop. The symbols in the figure are as follows. D and SI are data and scan inputs respectively. SE, scan enable is control line for the multiplexer to select one of D or SI. CP is the clock signal in both functional and test modes. D is used to capture the output response of the circuit and SI is used to scan in the test pattern.

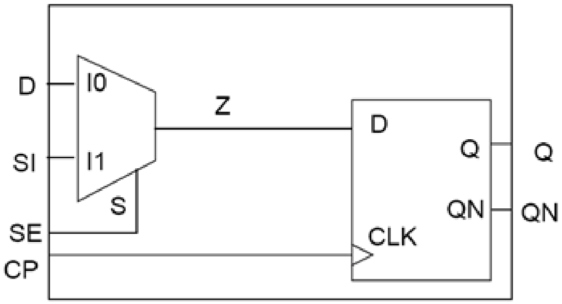


Figure 2. Muxed-D Scan Cell [49]

1.3.1.1. LSSD Scan

Figure 3 illustrates the design of a shift register latch (SRL) [16][17]. This level sensitive scan design (LSSD) cell is made of two latches L1 and L2. L1 is the master 2-port D latch and L2 is the slave. A, B and C are the clock signals. D and I are the data input and scan input ports respectively. Test requires the SRLs to be controlled by clock signal sequences. Single-latch design [16] or a double-latch design [18] can be used for LSSD for different clocking schemes.

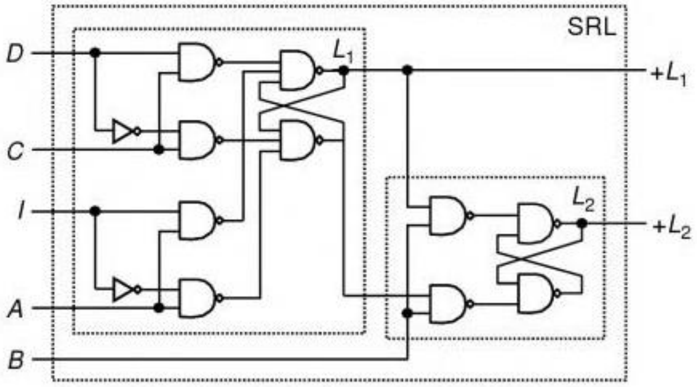


Figure 3. LSSD Scan Cell [49]

1.3.1.2. Enhanced Scan

Increasing the capacity of a scan cell to store two bits of data is enhanced scan [19][20]. The advantage of this is both the initialization vector and test vector can be loaded into the scan cell. If all the scan cells are flip-flops, a holding latch is added to each scan flip-flop at the output. Figure 4 illustrates the design of an enhanced scan cell. As you can see it needs extra area and extra power for additional circuitry. The two bits of the initialization and the test vectors are independent of each other. Unlike in the case of regular scan cell where the test vector needs to be scanned in or captured from circuit response and launched after the initialization vector, both of the bits are stored in place. This gives more control over the transitions that can be propagated increasing the fault coverage.

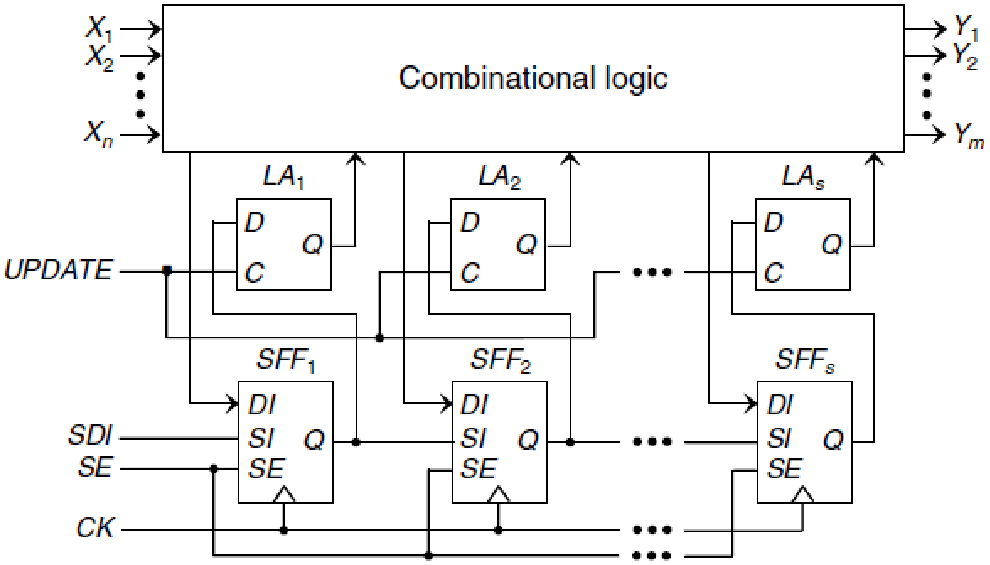


Figure 4. Example of Enhanced Scan [49]

1.3.2. Scan Based Delay Testing

As explained earlier in this section, two vectors, one to set the state of the line (0 or 1) and second to change the state (0->1 or 1->0) are required. Either if the initialization vector (A) is not able to set the state or the test vector (B) is not able to bring the transition at the target line the actual result would be different from the expected result at the corresponding observed point. Delay tests can use two types of clocking schemes, namely, Launch-On-Shift (LOS) [21][22] and Launch-On-Capture (LOC) [23]. Both of these schemes are illustrated in Figure 5. Prior results show LOS clocking scheme can achieve higher fault coverage. However, in LOS scheme, the last shift cycle and at-speed cycle needs to be properly timed for proper initialization of the circuit. It is complicated to design such high-speed scan design to activate the clock signal to all the scan cells within this narrow time.

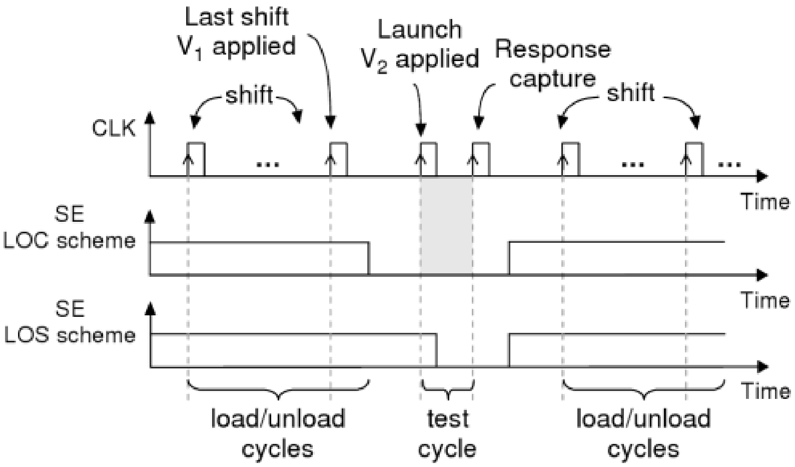


Figure 5. Clock Diagram for Scan Based Delay Testing [49]

1.3.2.1. Launch-On-Shift

In the Launch-On-Shift (LOS) scheme, the initialization vector A and test vector B differ by one-bit shift. When scan pattern is loaded into scan cells, the circuit is initialized. When the last bit of this scan pattern is shifted in to the scan cells, transition is launched. To capture the test response, an at-speed clock cycle is required. Switching off the Scan Enable clock signal to capture the response of circuit is crucial to the test here and it needs to be operating at at-speed. This demands two clock networks in the circuit and hence brings in additional timing constraints along with the regular clock. These drawbacks of LOS clocking scheme make it impractical to be used in high-speed designs.

1.3.2.2. Launch-On-Capture

In the Launch-On-Capture (LOC) scheme, two capture clocks are applied at speed to capture the test response into the scan cells. In this scenario, the second vector V_2 is the combinational circuit's response to the first vector V_1. The first capture clock is used to capture V_2 into the scan cells and launch transitions into the circuit, and the second capture clock is used to capture the test response. In LOC design, the SE signal is switched during the dead cycles between lowering the SE signal and applying the first capture clock, so it can operate at lower speed. As a result, the timing constraints on the SE signal are less aggressive, and hence LOC is used in high-speed designs.

1.4. Memory Model in Scan-based Delay Test

1.4.1. Black Box

Figure 6 shows how a memory is modeled as black box. In the scan based test, transitions cannot be propagated in and out of memory arrays in the circuit. While they are lost when entering the memory, only don't care (X) values are read out on the read paths of memory. The area around the memory thus creates a "shadow" region which cannot be tested. In this model fault coverage is reduced due to the lines present in shadow.

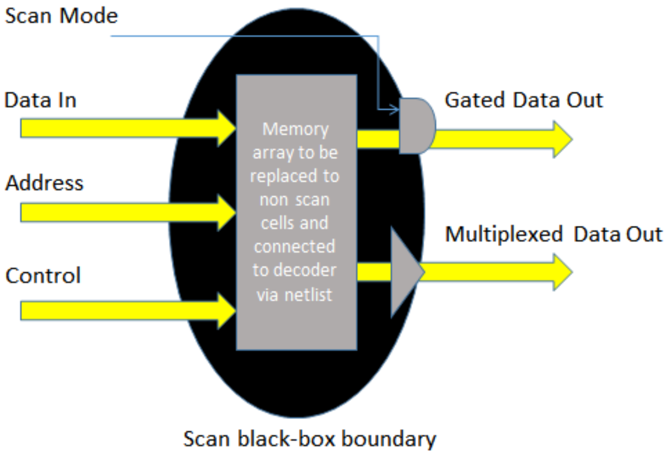


Figure 6. Memory Black Box Model

1.4.2. Memory Bypassing

The shadow regions can be eliminated by assigning values to the lines out of memory. The values on the read lines can be set statically or propagated directly from the input lines of the memory. This technique is depicted in Figure 7 and is known as memory bypassing. The unknown values can be stopped to be propagated to the circuit and all memory inputs can be captured. However, this assumes the memory is functional which means an exclusive test is required for the memories. The bypass logic could be inclusive of the some of the circuitry of a memory, for instance, the decoder logic. If so, partial delay test through memory is achieved. By selecting a particular word from the memory, bypass mode can also simulate power supply noise similar to functional operation.

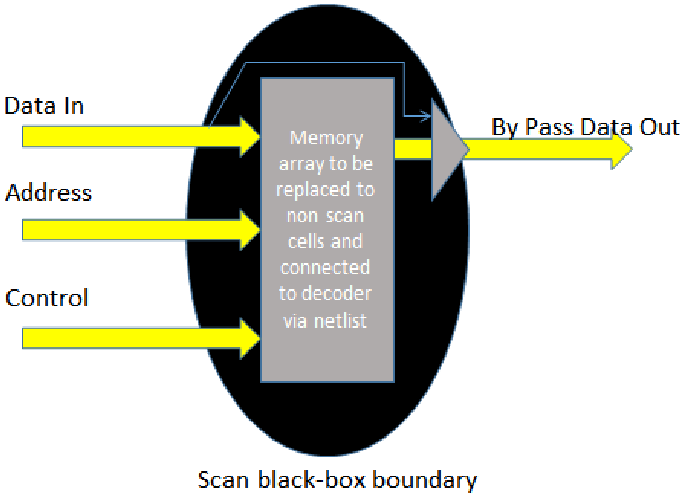

Data In
Address
Control
Memory array to be replaced to non scan cells and connected to decoder via netlist
By Pass Data Out
Scan black-box boundary

Figure 7. Bypassing Model

1.5. KLPG Algorithm and CodGen

CodGen is the in-house developed tool based on path delay fault model. CodGen is based on K Longest Path Per Gate (KLPG) algorithm [24][25]. K longest rising and falling paths are generated targeting each line in the circuit. This tool can be used for both combinational and sequential circuits. Considering only the longest path is not sufficient which testing the longest delay possible. This is because during fabrication, due to process variations, logically possible longest path may not yield the longest delay [26]. Figure 8 shows a probabilistic distribution of time taken by various paths. Let P_0 be the longest path followed by P1, P2 and so on. In the post-silicon design, P1 could be the longest path with delay defect of size greater than Δ_1. In our research, we have presented results for K=1 and K=2 for some heuristics and have observed not much change in the trend.

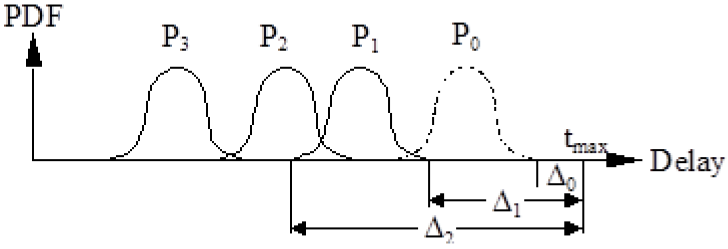


Figure 8. Probabilistic Distribution of Path Lengths

1.6. Pseudo Functional Test

In scan based delay test, the test vectors are launched at-speed not immediately after they are scanned in. The changes in supply voltage would significantly impact the accuracy of delay test [27][28]. The scan-in vector draws high current in short time from the supply leading to an inductance on the power grid. Figure 9 illustrates the inductive ringing in the circuit, caused by the sudden but temporary drop in the power supply voltage. The drop in supply voltage is marked on the left of Figure 9 which affects the functional speed of the circuit making it to operate slowly than normal. This is referred as *test overkill*.

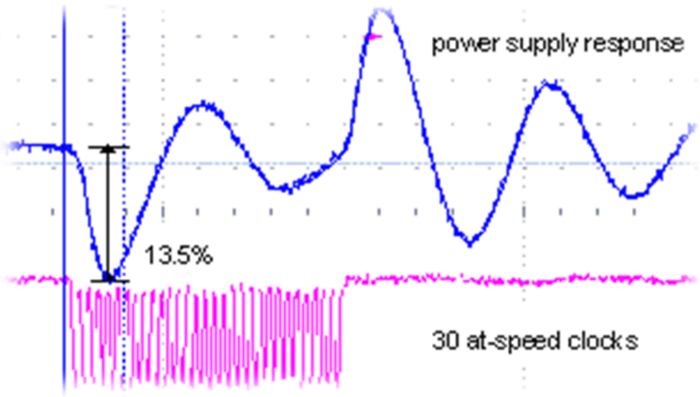


Figure 9. Delay Test Induces Drop of Power Supply Voltage [27]

Pseudo Function Test is the solution to this problem. The test vectors after being scanned in are delayed in applying to the circuit. A few *preamble* cycles which help the power supply noise to settle down are succeeded by test patterns before the launch at-speed cycles in the circuit as illustrated in Figure 10. Preamble cycles provide enough time to stabilize the drop in voltage. This makes sure delay test would not be affected. In sequential circuit testing, during preamble cycles, the scan cells capture the response of circuit and by the time of at-speed launch the test patterns are different from the scanned-in patterns. Scan chain cells are usually part of sequential memory to as it is costlier to have only scan cells and only non-scanned cells. Hence, *time frame expansion* is carried out to back-trace the necessary contents of scan cells before preamble cycles. The time frame expansion is in the number of preamble cycles used and test generator should be aware of this. Generating these test patterns through time frame expansion takes significant CPU time. The number of preamble cycles varies by the structure of the circuit.

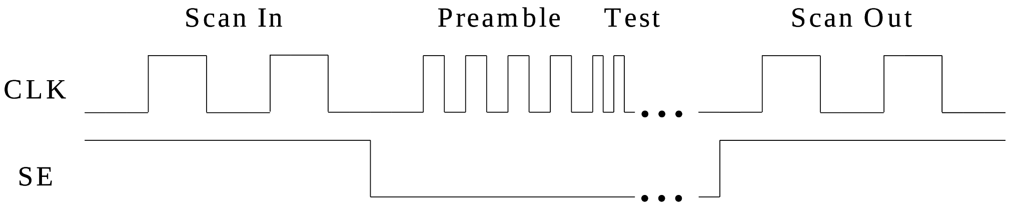


Figure 10. Clock Diagram of Pseudo Functional Test [44]

1.7. Structure of This Thesis

The rest of this thesis is structured as follows: In Section 2, the procedure for modeling and synthesis of structural memory models is presented. These models are needed to fit into the structural ATPG. The organization of CodGen is also explained. We present the test generation time taken for several benchmark circuits, which is the motivating factor for this research. Section 3 discusses several important data structures in *CodGen* and discusses the superlinear CPU time of the algorithm. Section 4 presents the reasoning behind this time complexity and compares various circuits to validate this analysis. A number of heuristics are evaluated to increase the performance of the algorithm, with experimental results given. Section 5 concludes with directions for future work, particularly the need for a major restructuring of the ATPG algorithms to achieve significant performance improvements.

2. MOTIVATION

2.1. Pseudo Functional Path Delay Test through Embedded Memory

System-on-Chip (SoC) boards are popularly built-in with memory arrays for efficiency. Functional March patterns [29][30] generated by memory built-in self-test (MBIST) [31][32][33][34] are used to test them. Scan tests are constituted of memory tests in case of Macrotest [36]. At-speed testing [35] of the components can be performed using Embedded Micro-Tester. With various technologies like the ones mentioned here, scan test can also be used to test latch-based embedded arrays [37]. The surrounding shadow regions around the memory can be tested using scan [39]. The non-scan memory cells on the board are tested using at-speed functional patterns [38].

DFT incorporated circuits can be thoroughly tested covering delay defaults in the memory. The fault coverage achieved using some of the techniques mentioned above is good. Our focus is to target small delay defects in embedded memories and try to optimize the testing process developed earlier [46] for larger memories. Prior research shows that some of the longest and critical paths in the circuit pass through embedded memories. An effective flow that can perform pseudo functional path delay test through embedded memory arrays was developed [46]. Figure 11 shows the possible paths through the memories. The read and write paths of memory adds significant number of gates to the circuit. The larger the memory the longer the paths tend to become as we include memory logic in the path generation. Testing these paths achieves good correlation between the maximum operating frequency (FMAX) of functional and structural delay test [29][30]. This also helps to reduce the defect levels substantially.

The algorithms and data structures developed in the prior research provide a straightforward solution to the embedded memory test problem, but they have proven economical only on small circuits. The time required to generate the tests increases superlinearly with memory size, contrary to the expectation of an approximately linear increase. This superlinear behavior is the bottle neck for scalability of the tool. Root causing the boot neck and exploring heuristics to overcome this has been the motivation for this research.

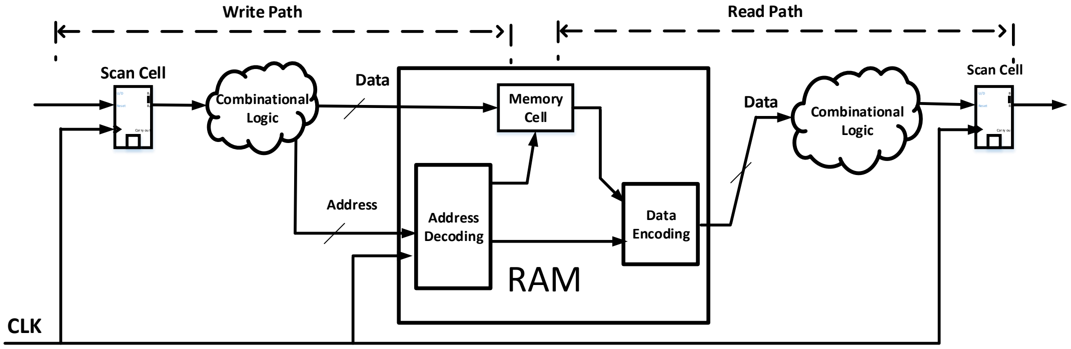


Figure 11. Paths into and out of Memory, reprinted with permission from [46]

2.2. Memory Arrays

Logic synthesis of the memory from behavioral Verilog model is achieved using some standard synthesis tools. This memory part of the circuit needs to be stitched back in with the rest of the circuit for ATPG. ATPG is extended to understand memory cells as flip-flops with multiplexers as depicted in Figure 12. These 2x1 multiplexers select between the new data or stored value in the flip-flop. The new data is written with write enable however the data in cell can be read out with the help of address decoder. Address decoder manages the control signal of the multiplexer based of the address supplied. The write to memory to a particular flip-flop activates the corresponding non scan cell using the address supplied. The read is performed from a memory cell similarly.

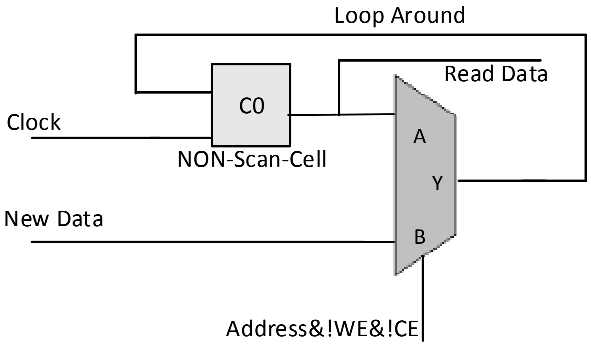


Figure 12. General Structure of Memory Array, reprinted with permission from [46]

A 4x3 memory array is synthesized in Figure 13. 12 non-scan cells are shown. It is composed of 2 address bits to select between one of the 4 words. 3 data bits are required to write into any one of the 4 address words at once. U53, U52, U56, U55, U61 and U58 are the series of AND-NOR gates forming the first layer of logic to select 6 possible bits out of 12 cells. The decoded address enables U53 and U52 to select one data bit out of the first four non-scan cells, each belonging to a different word. Memory array can be mathematical represented by a 2D array. In this case a 4x3 matrix. The first index is the word in the memory while the second index maps to the data bit. We have 3 data bits for each word. U53 and U52 give the value of matrix[x][0] (0^{th} bit in x^{th} word) where x ranges from 0 to 3. Other gates give the values of 1^{st} and 2^{nd} bits.

When this structure is integrated to the circuit, replacing the memory black box, scan cells are also added accordingly to input the required transitions and capture them at places other than the non-scan cells of the memory. The scan cells need to be placed in non-redundant fashion not to conflict with the transitions that can be launched from primary inputs or IO ports.

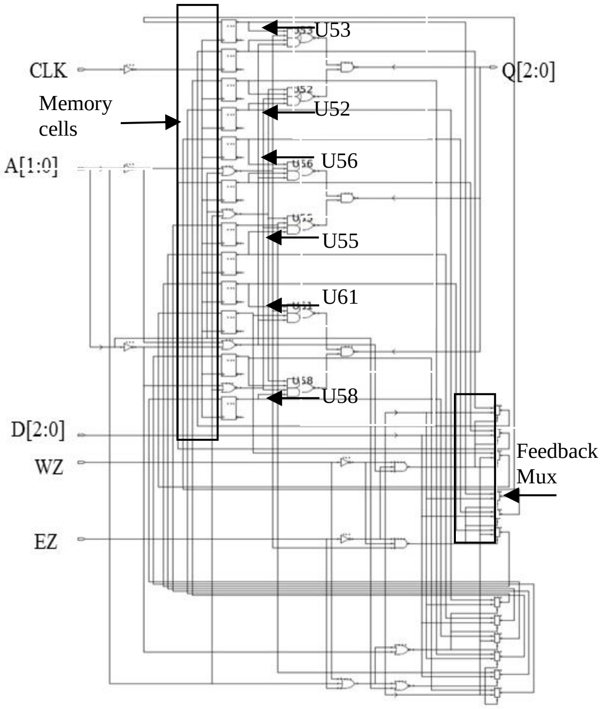


Figure 13. Logical Model of 4x3 Memory Array, reprinted with permission from [46]

The encoder consists of address and data lines going into the memory. All the lines targeted in this part of the circuit can propagate the transition to the memory or a scan cell input in this part of the circuit. If the value is captured in a non-scan cell, it must be propagated to a scan cell. If it is captured in the memory the readout occurs through the output decoder structure of the memory, such as shown in Figure 14. An important observation to note in memory circuits is there is a unique path from the each of the non-scan cells to the output of the memory. It passes through many gates. Hence, if one of the gates in this path is targeted for ATPG and a transition fault is not being able to propagate, there is intuitively less chance there would a successful propagation of transition for any other gate in the path unless it has other inputs in its fan-in cone. This correlation is not exploited in this research.

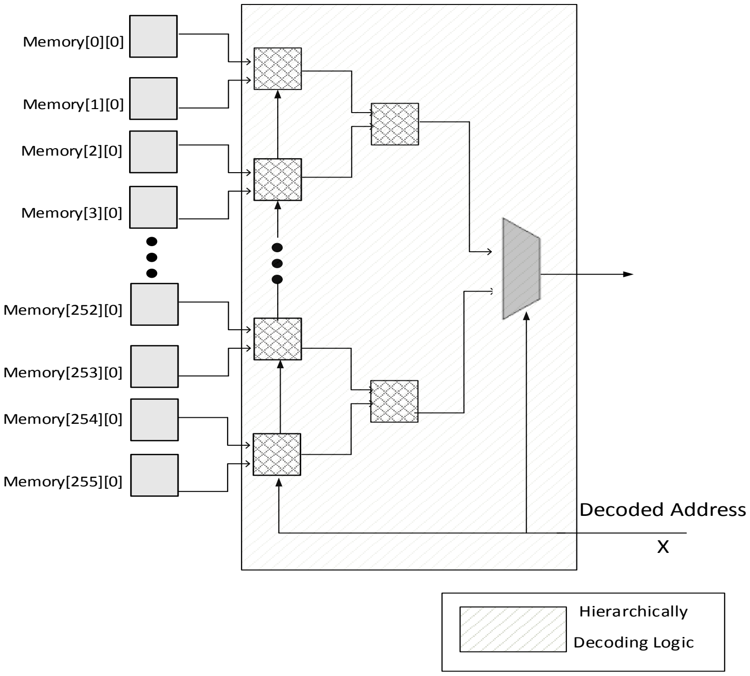


Figure 14. Output Decoding Structure in 256*8, reprinted with permission from [46]

2.2.1. PFT through Embedded Memory Test Generation Flow

Figure 15 outlines the procedure followed to modeled embedded memories for ATPG. The black box memory is synthesized to a structural “white box” memory model [46]. This mode is then integrated into the overall design. The entire circuit structure is later put through the phases of flattening, leveling and partitioning, producing two files. One file represents the levelized logic circuit and the other file describes the scan cells in the circuit. Levelization of the circuit helps in computation of a number of metrics, such as controllability, observability, and Esperance (longest path from a gate to an output) since the gates are stored in a sequential array and processed in the same way, which means they are processed in rank order.

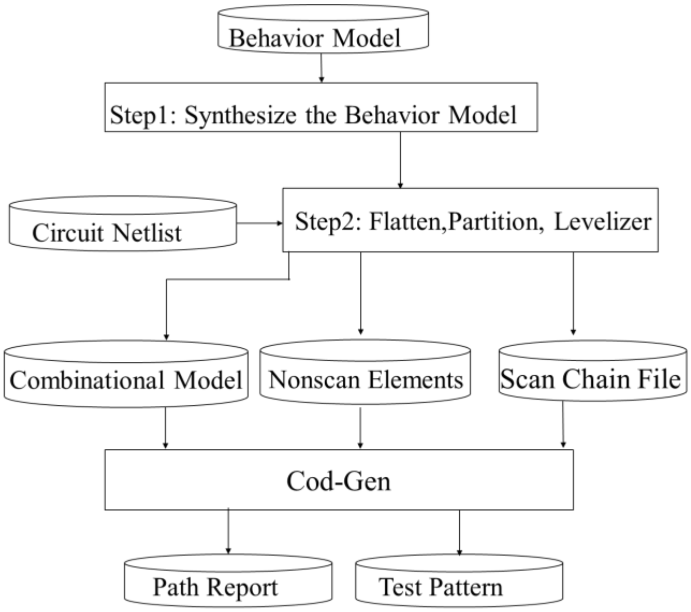


Figure 15. Flow of PFT KLPG, reprinted with permission from [46]

2.3. Difference in Launch of Transitions in Mux and Non-mux

In case of single mux gate, we have a 2x1 memory array with two cells, C0 and C1 and one data output as shown in Figure 16. The output is controlled by the address line and is chosen between C0 and C1.

Let C0 and C1 contain 0 and 1 respectively. A falling transaction is shown in the Figure 16. An address change from 1 to 0 is required to launch this transition. These values are written to these cells during the preamble cycles. Figure 17 shows a decomposed version of Figure 16. A and B are the lines from the memory cells. A falling transition is needed at the OR gate output. The control signal, S decides the signal to be propagated. The path grows from the NOT gate to AND1 and then to OR gate. Simulating a falling transition on S requires a steady 1 on line A. As we need to block interruptions from AND2 we set steady 0 on line B. These are the necessary assignments on the impacted lines of this transition. Setting to steady 1 of 0 avoids any glitches to propagate to observable point for a target line.

The length of the path in Figure 17 is 2 (length of path is given by the number of gates in it) more than the length of path in Figure 16. Imagine a circuit with larger memories where the length of path and necessary assignments increase in the number of gates added for conversion from mux to non mux.

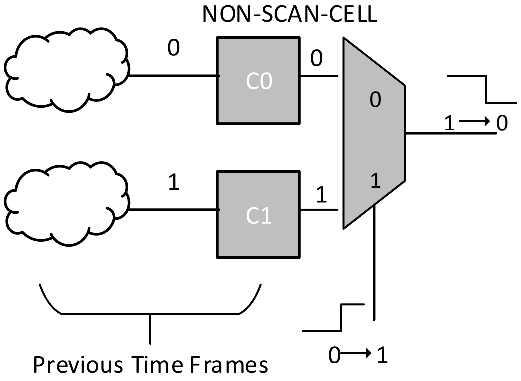


Figure 16. Launching a Transition by Toggling the Address, reprinted with permission from [46]

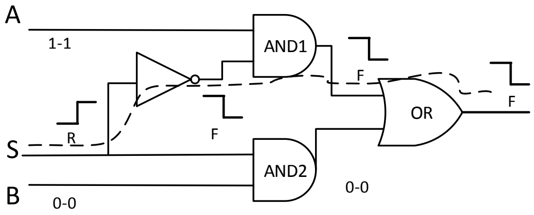


Figure 17. Necessary Assignments inside a Multiplexer to Propagate a Transition, reprinted with permission from [46]

The assignments in the memory cells are inferred from the direct implications. A sufficient number of preamble cycles are required to write all of the necessary values to the non-scan cells. In our example since we are using 2 values out of 2 different cells, we need two preamble cycles to write these values, assuming only one word can be accessed at a time. No constraints are applied on the address and data inputs to the memory. The target lines farther from the memory are impacted by more side inputs and more non scan cells. It is advisable to choose the number of preamble cycles and values in memory in such a way to have minimum impact on the target line. The direct implications also restrict side inputs required for propagation which are justified later.

The timing of a Pseudo Functional KLPG (PKLPG) test with *coda* cycles is shown in Figure 18. The scan pattern shift in and shift out is done in 4 slow cycles. We have 4 preambles, 2 at-speed launch and capture cycles, and 4 coda cycles for propagation.

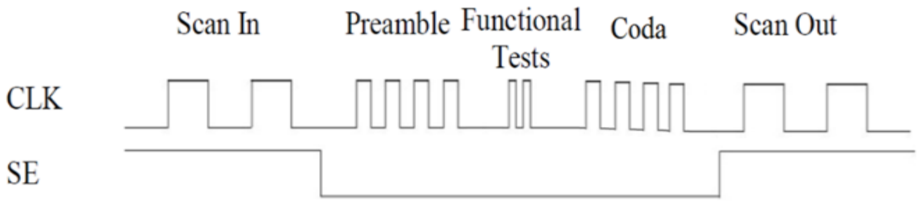


Figure 18. Coda Cycles in PFT KLPG, reprinted with permission from [46]

2.4. CodGen Modules and Functions

The flow chart of CodGen is shown in Figure 19. The steps in the flow are described in more detail in the following sub-sections.

2.4.1. Pre-Processing

Pre-processing consists of reading the gates and scan cells and linking them appropriately. Calculation of SCOAP metrics for controllability and observability is done in this stage. For each of the gate lines that is being targeted, we generate fan-in and fan-out cones to understand the primary inputs (PIs) or pseudo primary inputs (PPIs) that could influence the transition on the target line. Both rising and falling partial paths are created, since the transition on the line could be produced by either or just one of them.

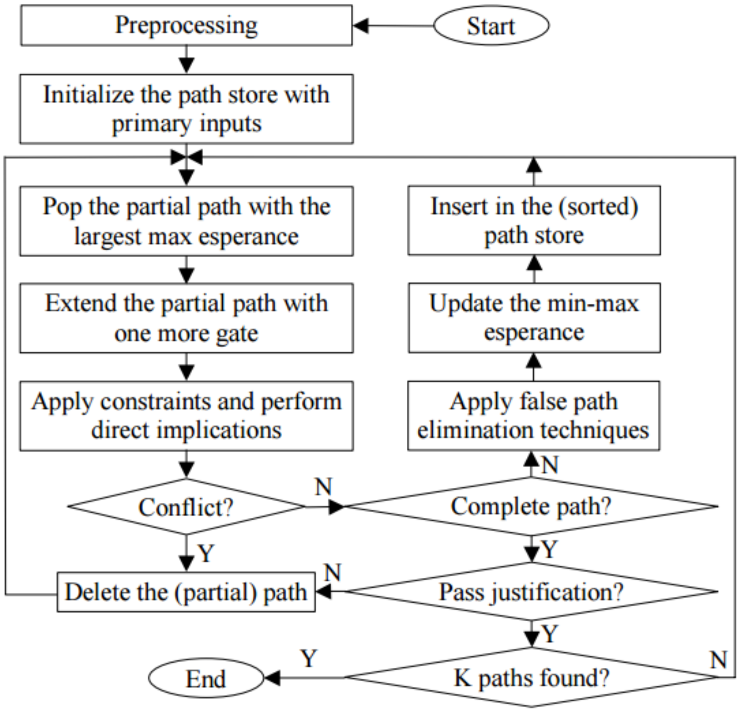

Preprocessing
Start
Initialize the path store with primary inputs
Pop the partial path with the largest max esperance
Extend the partial path with one more gate
Apply constraints and perform direct implications
Conflict?
N
Y
Delete the (partial) path
Insert in the (sorted) path store
Update the min-max esperance
Apply false path elimination techniques
N
Complete path?
Y
N
Pass justification?
Y
End
Y
K paths found?
N

Figure 19. CodGen Flow, reprinted with permission from [24]

2.4.2. Path Generation

Path generation is the crucial part of the CodGen algorithm and the focus of our research. The partial path store is initialized with the paths that are one gate in length, starting from all the PIs and PPIs in the fan-in code of the target line. The paths are sorted by Esperance (French for "hope"), the upper bound of the length of the complete paths (that reach a primary output or pseudo primary output) that start at the end of each partial path. A path is grown one gate at a time, with assignments on the side inputs based on the sensitization criterion. Direct implications are performed based on those assignments. If a conflict is found, the partial path is discarded as a false path.

The path generation is related to Dijkstra’s Algorithm in being greedy in the selection of the next node required in the propagation of a transition and that completes the path. However, the criteria is much different in the sense, we are selecting a better path every time instead of the node to we want to extend the path. Also, the time taken for sorting of the new edges in Dijkstra’s algorithm is best achieved using a Heap structure (any structure which would efficiently present the nodes in sorted order). We observe this intuition is not completely true with our paths and we have presented the reason for it in section 3.

The path generation algorithm is described in Figure 20(a) and 20(b). The following can be considered the major modules of the code in the order of their serialization.

1. Reading in the circuit and scan chains (done once)
2. Initialization of metrics and circuit data structures for the target line (once per target)
3. Finding fan-in and fan-out cones for each gate (done once per target)
4. Path Generation (once per path)
5. Final Justification (once per path if it succeeds)
6. Dynamic Compaction of the path (once per path)

Prerequisites:
I. SCOAP metrics are calculated.
II. Gate delays added to data structures and Esperance for all the gates updated recursively from out most layer
III. Circuit initialized to hold Present Values
IV. The K longest paths for the target fan out for the gate have not been achieved yet.

//The following are the essential macros for path generation:
#define MAXTRY a
#define MAX_SUCCESS_PATHS b
#define K
#define MAX_PARTIALPATH_POOL d

Figure 20 (a). Path Generation Algorithm Prerequisites and Macros

```
Path_Generation() :
    for gate g in Gates of Circuit(G) {
   for fanout f in fanout gates of g(F):
          estimate fan-in cone
          estimate fan-out cone
          partial path pool initialized from PIs and PPIs in fan-in cone
         while(TRUE) {
         1.if K rising and falling paths found
                  return
         2.if MAX_TRY expansions done
                 return // how many times we extend the paths in this partial store
         3.if MAX_SUCCESS_PATHS found
                  return //For this fan out we already found MAX_SUCCESS_PATHs
                           //but not able to validate K rising and falling paths

         4.Retrieve the partial path P' with next highest Esperance from the partial
                 path pool
         5. AnyCompletePath_found = Expand(P') ;
            Expansion_path_tries_for_this_fanout ++;
         // This is where are expanding the path to all of its fanouts.
         // In case none of  the fan outs are valid, we would be dropping the path here
         6. if (AnyCompletePath_found) { // complete path p be found
                 Complete_path_tries_for_this_fanout ++;
                  If (p passes the justification) {
                              Update_Coverage();
                              Record_path();
                              Process p through Compaction for test vectors generation;
                  }
         }
         7. Limit the partial path pool size to MAX_PARTIALPATH_POOL
        }// Closure of while

   }// Closure of processing of fanout f of gate g
```

Figure 20 (b). Path Generation Algorithm

Upper bounds are set to limit the time spent in partial path pool search. There is a limit on number of times we try to extend a partial path before giving up. It is currently set to 10000. This has proven sufficient for circuits with 200,000 gates or more. The fault coverage saturates with few paths hitting the “max try” limit. For a 2048x8 memory, there were 12 million path extensions for the entire circuit, with the average number of extension attempts per path of close to 60. A few paths did reach the extension limit. Extensions are not carried out on all gates, since many gates are not of interest. For example, gates with a single fan-out do not need to be processed by the extension code, since the extension is guaranteed (e.g. a string of buffers).

2.5. Coverage Update for the Path Gates

A longest path through a gate could be the longest path through many of the other gates on this path [47]. We keep track of the longest paths found through each gate until now and update this as each new complete justified path is found. This is exploited during path generation as explained in [42]. K longest paths for each gate are stored in containers $L_{ub}[1 \ldots K]$ and $L_{lb}[1 \ldots K]$ in descending order for the upper and lower bounds of lengths respectively. Upper bound for each gate is initialized to the longest structural path through the circuit whereas lower bound is initialized to 0. In the path generation process, the upper bound path lengths decrease and lower bound path lengths increase to the actual path lengths found. When processing of gate g_i is completed these containers will be updated to the actual lengths of K longest testable paths. The idea behind updating the coverage is as follows. A longest path through a gate g_i contains many gates. Each of these gates, g_j lower bound lengths container $L_{lb}[1 \ldots K]$ will be updated with the newly found path length if it is greater than the least of them. Figure 21 presents an example with a complete path. In this case, K is 3. In path generation, till a certain point of time, let us say the lengths of 3 longest paths through g_j were found to be 22, 18 and 15. However, when generating paths targeting the gate g_i, a new path of length 20 through g_j is found. Hence, the L_{lb} of g_j is updated to {22,20,18}. Similarly all the gates in the path are updated.

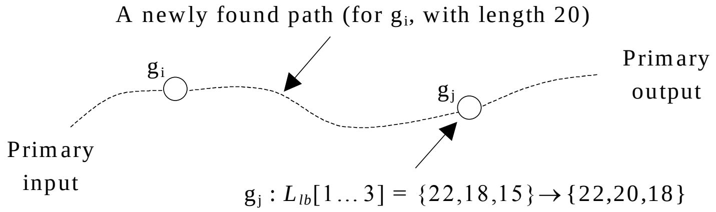

A newly found path (for g_i, with length 20)
g_i
g_j
Primary output
Primary input
$g_j : L_{lb}[1\ldots3] = \{22,18,15\} \rightarrow \{22,20,18\}$

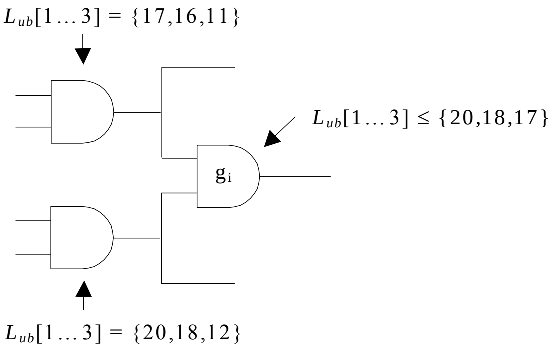
Figure 21. Updating Llb[1…K], reprinted with permission from [42]

The upper bound path lengths for a gate g_i, $L_{ub}[1\ldots K]$ can be used to calculate the upper bound path lengths of its fan out gates. Suppose g_i has n fan-in gates and let U indicate the union set of L_{ub} containers of all of its fan-in gates. The maximum lengths of K longest testable paths through g_i cannot be more than the K maximum elements of U. This is because all the paths through g_i must be extended from one of its fan-in gates. Also, optimistically, in level wise processing of path generation of gates, all the g_i's fan-in gates K longest testable paths are generated before looking at g_i. Figure 22 presents an example for the upper bound path lengths. With K=3, let us say g_i has 2 fan-in gates with L_{ub} values {17,16,11} and {20,18,12}. This means L_{ub} for g_i must be {20,18,17}. Similar analysis can be performed for absolute denominators [48] for gate g_i.

The intuition here is as the path generation process targets more and more gates, the values of L_{lb} and L_{ub} for gate g_i, which have not yet been targeted for path generation, get close to each other. If the values are close enough (how close needs to defined prior, like less than 1% difference), it means they represent the K longest testable path lengths through g_i

and path generation for g_i can be skipped. Many gates can be skipped using unit delay model [47].

If we have to generate paths for gate g_i and max Esperance of the partial path being processed is less than L_{lb}[v](1<v<k), then none of the partial paths in path store have a chance to be longer than L_{lb}[v]. This means that the first v paths in L_{lb} are v longest paths through gi, and hence are updated in first v positions in L_{ub} respectively. Also, if min Esperance of the partial path is greater than L_{ub}[v], during the propagation of vth longest path, it can be deleted since; it is either a false path or will grow to be a path already found.



$L_{ub}[1\ldots3] = \{17,16,11\}$
$L_{ub}[1\ldots3] \leq \{20,18,17\}$
g_i
$L_{ub}[1\ldots3] = \{20,18,12\}$

Figure 22. Updating Lub[1…K], reprinted with permission from [42]

Let the highest value in L_{ub}, Z is the length of longest possible path though g_i so far. If a path targeting one of the other gates, g_j grows to the gate g_i and has maximum Esperance greater than Z, it must be reduced to Z. A complete path passing though g_i cannot have a length greater than Z. During the path propagation, if min Esperance becomes larger than max Esperance, this partial path eventually grows to be a false path, and hence deleted.

2.6. CodGen with Embedded Memory Array

CodGen is capable of generating the longest paths through the circuit if the memory is modeled as flip-flops and decoder as combinational logic. Longest paths through memory arrays can be divided into two types: longest paths into memory arrays and longest paths out of memory arrays. The longest paths into the memory are generated for the gates in the write path. The path starts from the input ports of the memory. In this case, the path result is captured in a non-scan memory cell. The captured Boolean values are propagated to a scan cell using extra *coda* clock cycles. The low-speed coda cycles are not timed, and select the easiest path to propagate to a scan cell. Potentially several coda cycles are needed. Currently we only use one coda cycle.

Testing the longest paths out of the memory requires launching a transition at the memory output, on the read path shown in Figures 16 and 17. We are able to launch the transition by assuming these values could be set into the memory cells in the given preamble cycles. Currently two preamble cycles are sufficient to write into two memory words. More preamble cycles might be required if there are additional sequential constraints.

2.7. Cost of Various Circuits Using CodGen

CodGen is used to test various circuits with embedded memory arrays and the time taken for path generation for these circuits' increases in accordance to unknown factors. Table 1 shows the total time taken for test generation for the circuits. The notation "mux" indicates memories that use a multiplexer primitive in the logic, while "non_mux" means the mux is replaced by gates as in Figure 17. In the first two circuits, the 256x8 bit memory is modeled structurally, while in the last two circuits, the 2048x8 bit memory is modeled.

Table 1. Total Time Taken by CodGen on Various Circuits

Circuit	No. Gates	No. Target Lines	CPU Time (DD:HH:MM:SS)
STC_1_256_8_mux	47777	101129	0:01:12:54
STC_1_256_8_non_mux	56206	113907	0:03:16:09
STC_1_2048_8_mux	92675	221473	0:03:27:31
STC_1_2048_8_non_mux	160246	323847	1:10:35:32

In attempting to estimate and model the time complexity for the circuits, the following questions are raised:

1. With the increase in number of gates (due to modeling the memory as gates), there is a linear increase in the number of target lines; therefore the time taken for the path generation should be roughly linear as well, since the path generation for each target line is a separate process. However, this is not the case in Table 1. For the largest circuit, we can see that the gate count is at least 1.5x that of the other circuits, but the increase in CPU time is 7-8x.
2. Between the mux and non-mux designs, the increase in gates is only due to the extra gates (2 ANDs, an OR and a NOT) to represent the MUX gates. The circuit depth increases as well in terms of number of gates on the longest paths. The CPU time increases much more than the corresponding increase in gate count or path length. For path lengths of N gates, the CPU time should go as O(N·log$N$·log$N$) since it takes N expansions, and each expansion requires inserting the partial path in the path pool, which takes O(log$N$) time, and at each expansion it involves identifying the fan-out with longest Esperance, which could also take O(log$N$) time.
3. To get a better understanding of which circuit is standing out from the specified behavior Figure 23 is shown for comparison. With this, we understand further

analysis is need for the time consumed by various modules of CodGen. It is possible that all of the modules are taking correspondingly more time, or one or a small number of modules have a high time complexity. This is analyzed in detail in Section 3.

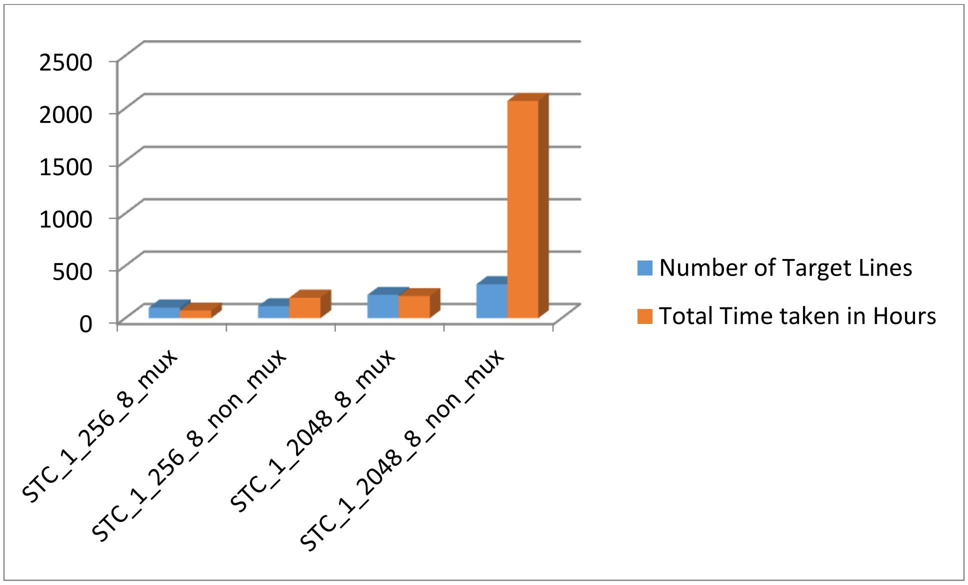


Figure 23. Total Time Taken by CodGen on Various Circuits

3. TIME COMPLEXITY OF CODGEN

3.1. Time Profiling of Various Circuits in CodGen

We have divided up the CodGen algorithm into the following modules. Each of these are independent of the other as described in Figure 20(b).

1. Initialization time
2. Fan in Cone and Fan out cone finding time
3. Path Generation Time
4. Justification Time
5. Compaction Time

Table 2 lists the CPU time (in seconds) for the modules for the benchmark circuits. The following sections and figures discuss path generation, justification and dynamic compaction time in more detail.

Table 2. Time Profiling of Various Circuits in the Identified Modules

Circuit	**Initialization**	**Fan-in and Fan-out Cones**	**Path Generation**	**Justification**	**Dynamic Compaction**
STC_1_256_8_mux	14.52	1.008	197.411	1316.42	2648.36
STC_1_256_8_non_mux	69.03	1.524	292.404	3790.16	7352.65
STC_1_2048_8_mux	15.616	3.585	1169.16	4384.92	4331.78
STC_1_2048_8_non_mux	35.168	86.251	31790.5	24215.7	66992.1

3.1.1. Path Generation

Figure 24 is the comparison of the time spent in path generation. Comparing 256x8 non-mux to 2048x8 non-mux, we see a 109x increase in CPU time for an 8x larger circuit. This is surprising and calls for in-depth analysis. We can also see the comparison between 256 mux and 2048 mux. There is a 6x increase in CPU time for an 8x increase in circuit size, which is quite reasonable. For the non-mux circuit, we might consider some super linear increase in time due to a few extra path generation failures or the deeper circuit. However, the increase in time for 256x8 mux to 256x8 non-mux is 1.5x. The increase in time taken for 2048x8 mux to 2048x8 non-mux is 27x. This is huge increase considering the gate count increased by only 47%. In the rest of this work, we analyze and observe the behaviors of the 256x8 non-mux and 2048x8 non-mux circuits closely.

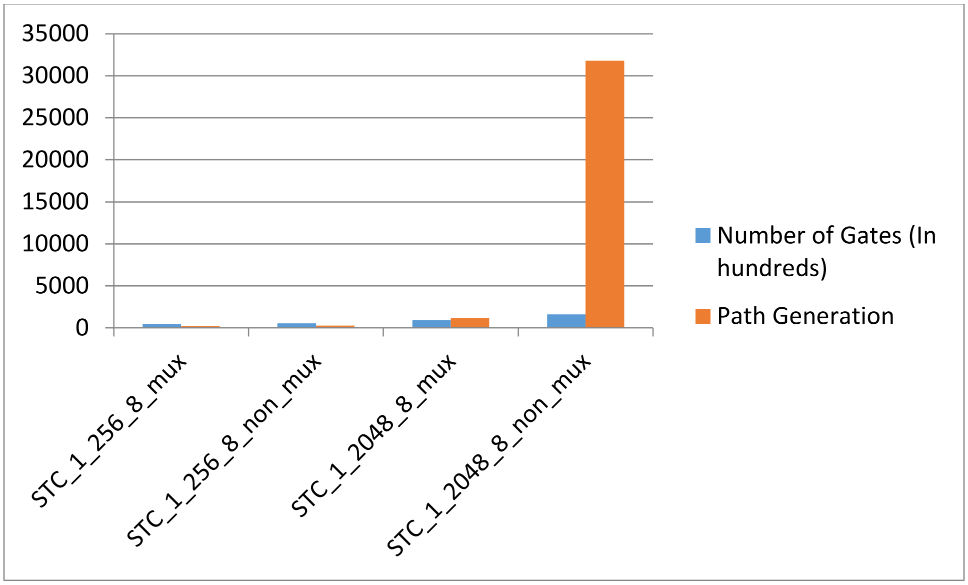


Figure 24. Path Generation Module Cost Comparison

3.1.2. Justification Time

The justification time (in seconds) for the benchmarks is shown in Figure 25. The time increases by 5.5x going from the 2048x8 mux to 2048x8 non-mux designs. This calls for a look at the path lengths and number of paths that are being justified. The justification is done by the MiniSAT SAT package, so it is not a part of the algorithm in Figure 20(b). We present a reasonable hypothesis for this increase in section 3.3 of this thesis. However, we believe this module needs to be explored further for proper mathematical reasoning.

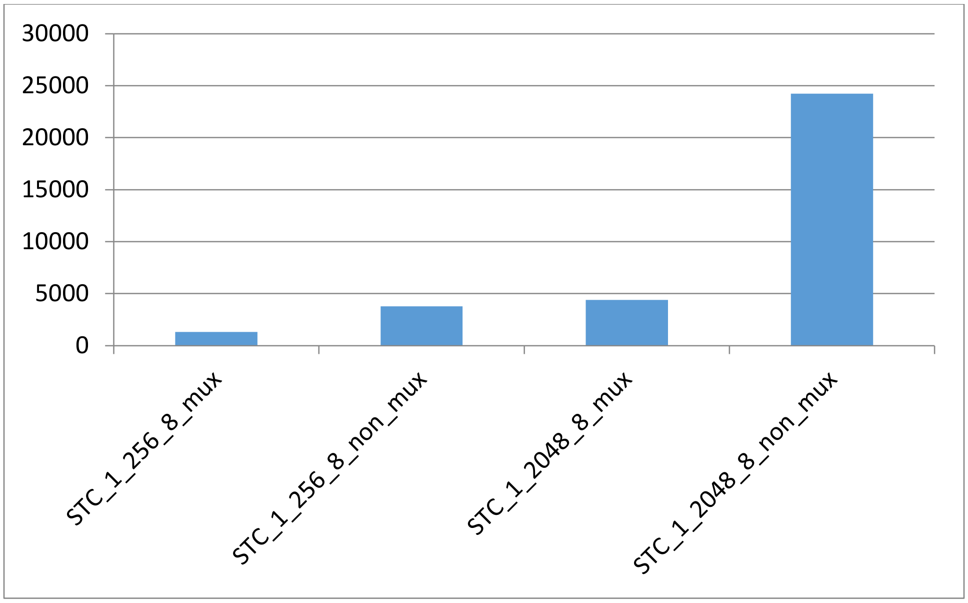


Figure 25. Justification Module Cost Comparison

3.1.3. Dynamic Compaction

The dynamic compaction happens in quadratic time since each new test may need to be compared with all the patterns in the pattern pool to identify a pattern compatible with the test. However, since the patterns for the mux and non-mux versions of the designs should be the same, this CPU time should be the same. The time difference must come from the fact that when a test passes the initial compatibility test with a pattern, a justification is run to verify this, using MiniSAT. As with Justification, there is a large increase in CPU time going from the mux to non-mux versions of the design. Table 5 presents the cost comparison for dynamic compaction for all the circuits. In section 3.4, we present a reasonable hypothesis trying to explain the superlinear increase in compaction time.

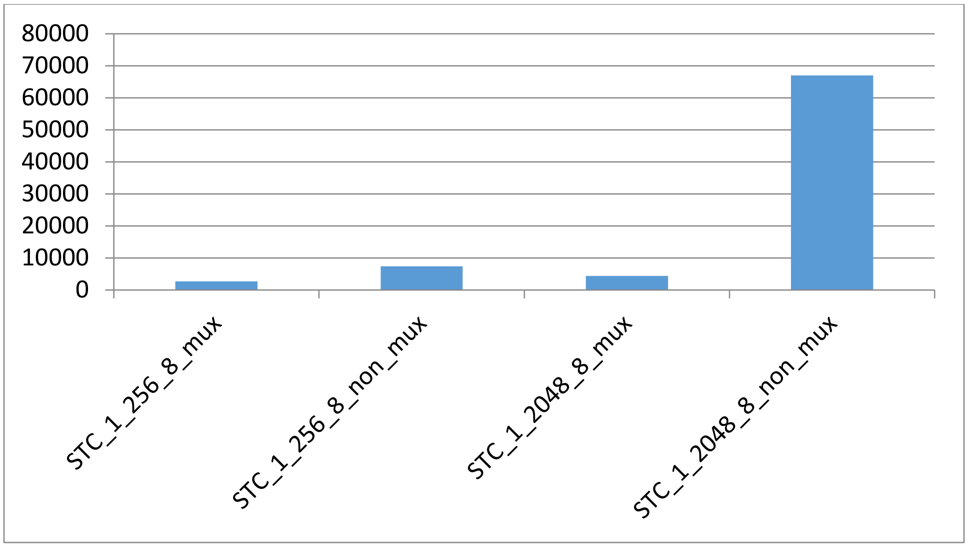


Figure 26. Dynamic Compaction Cost Comparison

3.2. Path Generation Internal Modules

To further understand the time complexity of path generation, the internal modules of path generation are studied. The following are the influencing factors for the superlinear increase in CPU time. We study the 2048x8 non-mux path generation time relative to the 256x8 non-mux path generation time.

3.2.1. Partial Paths per Circuit (Factor N1)

Each partial path is formed by adding a gate to its parent. It is processed only once. If a path is complete, but the value is not captured in a non-scan cell, it needs to be propagated to a scan cell. In either case, the extension and operations applied on the partial path are the same except for the additional direct implications if we are still propagating. Figure 27 gives an idea of how partial paths are created. If a gate's fan-out is *N* there would be *N* partial paths added to the partial path store. This means the total number of iterations of the algorithm is dependent on number of partial paths processed, which might be proportional to the average number of fan-outs.

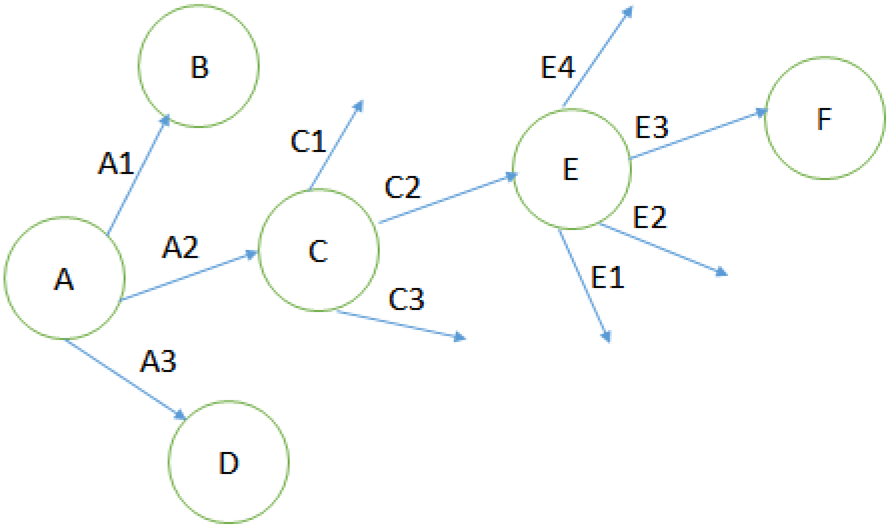


Figure 27. Partial Paths Formed

Figure 28, we show a graph to compare the increase in partial paths to increase in fan out for all the circuits. We subtracted the number of scan POs from total gates from which a partial path cannot be generated. It can be seen that the number of partial paths created is not linear with the increase in fan-out.

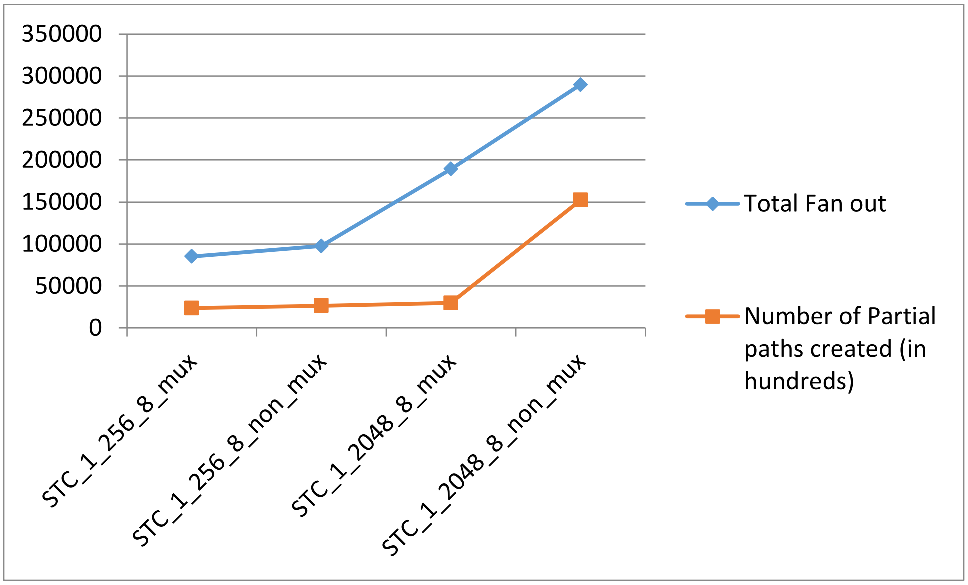


Figure 28. Fan-out vs Partial Paths Created

The deviation in behavior is only for the 2048x8 non-mux circuit. We will estimate this factor as the ratio of total no. path extensions in 2048x8 non-mux to total no. path extensions in 256x8 non-mux (15550967/ 2580858). **Factor N1 = 5.**

3.2.2. Partial Path Processing (Factor N2)

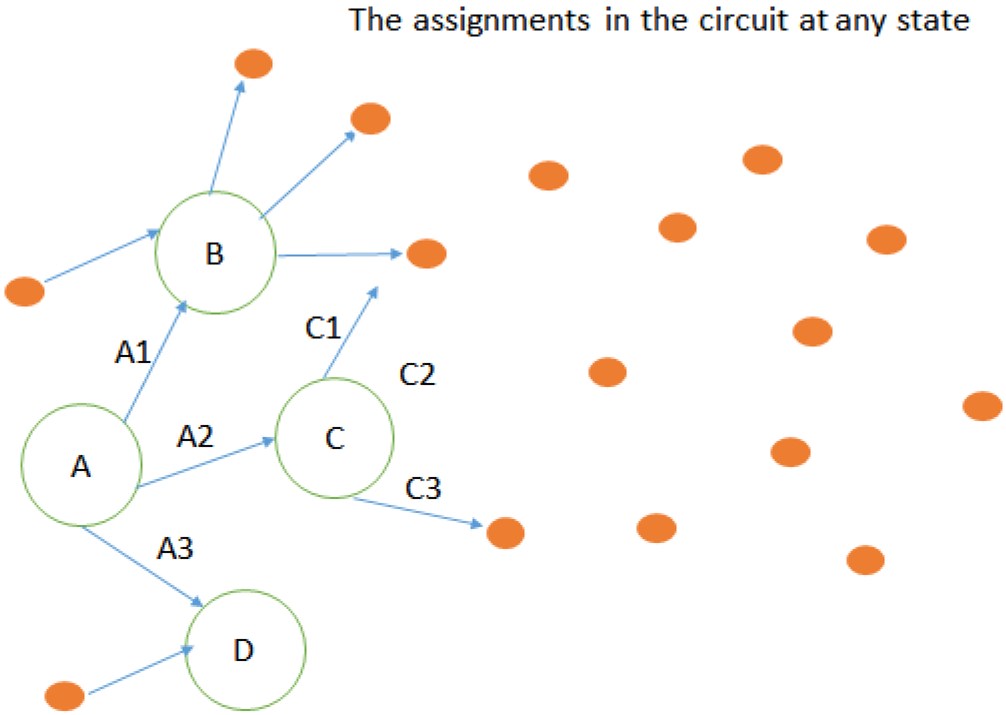


Figure 29. Necessary Assignments for Path Extension

For each path extension, the circuit needs to brought back to the initial state with all the assignments of its parent. In Figure 29, when A1, A2 and A3 are extended from A, each of them have separate assignments based on the type of gate B, C and D respectively. If A2 has the maximum Esperance, it would be chosen and extended to C1, C2, and C3. During this extension, assignments made on A2 will be carried to all of fan-out paths. So before extension, the circuit would be reconstructed to the level of its parent. Also note that the circuit after the extensions needs to be reset to its initial state. The time complexity would be in the order of number of assignments made on each partial path. The number of additional logic levels between the two circuits (2048 non mux, 256 non mux) is $\log_2(2048/256) = 3$. **Factor N2 = 3.**

3.2.3. Direct Implications (Factor N3)

When we are comparing two circuits, the time for direct implications (DI) is on the order of the increase in the depth of the circuit. The assignments are not duplicated and are computed on the fly. This means if a mux is replaced by 2 ANDs, an OR and a NOT gate, we cannot derive the assignments with the pre-knowledge of this mux to non-mux transformation. A signal transition flowing through these gates would need three extensions of the partial path and each extension has additional assignments. The non-mux circuit also has additional target lines. Hence, the number of assignments would go at least in the order of increase in circuit depth. But the number of times these would be performed would be directly proportional to the overall average length of the path.

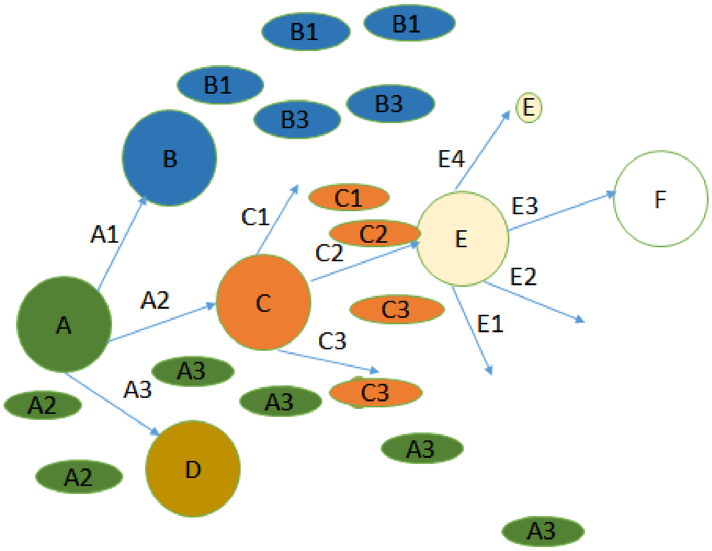


Figure 30. Assignments of Partial Paths

Figure 30 shows the data stored in the data structures. The direct implications on A1 expansion would be performed during the time of expansion of A1 and similarly for A2 and A3. Now these would be part of the assignments made on partial paths A1, A2 and A3 respectively. However, instead of duplicating the assignments for all the 3 paths, a parent path pool is used to store partial path A. Now all the assignments obtained in A would be associated with that parent path. Similarly, if a path is extended to F as shown in Figure 30, the additional assignments acquired with direct implications would only be allocated to F. It would have the parent paths as E->C->A whose assignments would be borrowed as necessary whenever we pop out a path created out of a fan-out from F.

Table 3. Direct Implication Time

Circuit	**Direct Implication Time(sec)**
STC_1_256_8_mux	99.39
STC_1_256_8_non_mux	157.821
STC_1_2048_8_mux	1029.44
STC_1_2048_8_non_mux	7651.25

From Table 3 we can observe that the time taken is not completely proportional to the increase in structural depth or increase in average depth of the circuit. This is because; from 256x8 mux to non-mux the number of gates changed (MUX being replaced with ANDs, ORs, and NOTs) is not proportional to the total number of gates in the circuit. However from 2048x8 mux to non-mux there is a much larger impact on gate count. This is why the latter circuits have a CPU time increase closer to the complexity estimate. Direct implications contribute to factor N3. **Factor N3 is (N1)(average path length)(N2) = (5)(2.8)(3) ~ 40.**

The work based on the **average path length** is described in the next section.

3.2.4. Average Path Length (Factor N4)

When we are processing a path of length *N*, the total paths that would have been processed have lengths *N*-1, *N*-2, *N*-3…1. The operations performed to reach this point would be on the order of the current path length **N**. In Figure 30, a typical path growth in the circuit can be seen. If extending from A which is of length 1(one gate, may be primary input or scan primary input), A1, A2, A3 would be of length 2 and C1, C2, C3 would be of length 3 and so on. Now, the estimated time for generation of these paths can be analyzed as follows.

A1 => reconstruction (A) + destruction (A) + DI(A1) + D1(A2) + DI(A3)

Reconstruction is necessary assignments that the gate in A has brought in the circuit. A single circuit instance with 0 value assignments for all the gates is initialized and with every partial path popped out, it is instantiated with assignments of all its parent paths. Deconstruction refers to clean up of circuit. Let reconstruction and destruction be estimated as const() time.

It is important to note that no matter which we chose among A1, A2 and A3, the work required for all of the partial paths needs to be done. The above equation transforms to

⇨ Const(A) + DI(A1) + D1(A2) + DI(A3)

Similarly the required work for C1 is

⇨ Const(C) + DI(A1) + D1(C2) + DI(C3)

Figure 30 also shows the assignments of each gate in the circuit. Each assignment is an oval with path name written inside it. Consider paths A2 and E3 of length 2 and 4 respectively (length is the number of gates on the path).

Work done for A2 = W(A2)

= Const(A) + DI(A1) + DI(A2) +DI(A3)

= Const(A) + 3*DI(A1)

=Const(A) + 3* di => Let di be the average number of assignments made for a gate

= Const(A) + DI => **Equation 1.**

Let DI be the average number of assignments made on the extension of any gate in the circuit.

Work done of E3 = W(E3)

= Const(E) + DI(E1)+DI(E2)+DI(E3)

= Const(C) + DI(C1)+DI(C2)+DI(C3) +3*DI(E1)

Approximately, at this level all fan-outs might take the same time for their direct implications

= W(A2) + 3*DI(C1) + 3*DI(E1)

= W(A2) + 2*(3*di)

= W(A2) + 2*(DI) => **Equation 2.**

For long paths W(A2) in Equation 2 would be negligible, since the starting point is always one gate in our algorithm. Also, note that W(A2) will be more than or equal to Const(A). Hence, from any starting point, A, equation 1 and 2 show that the work done on a path would be at least the number of times a path is larger than any other path. Factor **N4 (Average Path length of 2048 non mux/Average Path Length of 256 non mux) = 2.8.**

3.3. Path Justification

The paths found in the Path Generation are topologically longest paths, if there are no false paths. In spite of the assignments and direct implications made to the circuit for the propagation of this path, some complete paths can be a false path, which means that the transition cannot propagate along the path. Because of this the paths must be justified to check whether they are sensitizable. During justification, a set of input value assignments are found to set the necessary assignments. An example of a sensitizable path is shown in Figure 31. In this example, *a=0* and *c=1* are necessary assignments to sensitize the path from *b* to *g*, and *X0* and *S1* are the values on *a* and *c* to justify them. Here, “X0” on *a* means that the value is “X” or “don’t care” (DC) in the first vector and 0 in the second vector. The value “S1” on *c* means “stable one”.

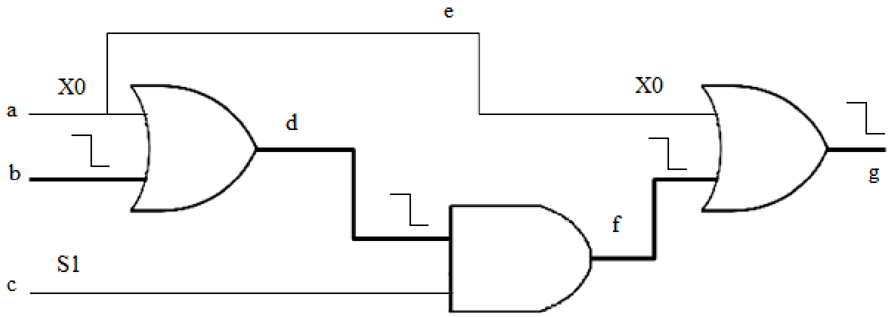


Figure 31. Example of Sensitizable Path

Justification time is proportional to two factors.

1. Length of the path.
2. The number of paths justified

Table 4. Number of Paths vs Justification Time

Circuit	No. Paths (Final unique paths/ total paths)	Justification Time (sec)
STC_1_256_8_mux	8854/19374	1316.42
STC_1_256_8_non_mux	13623/43557	3790.16
STC_1_2048_8_mux	10192/33372	4384.92
STC_1_2048_8_non_mux	25208/85862	24215.7

The number of gates in from 256 non mux to 2048 non mux increases by **3** times (Table 1). Number of paths justified for 2048 non mux to 256 non mux = 25208/13623 ~ **2**. Both of these factors are directly proportional to the CPU time. Hence, the CPU time for the external SAT engine, considering linear time operation would increase by (3).(2) = 6 times. We see the increase in time from Table 4 is close to 6x, which is almost as expected.

3.4. Dynamic Compaction

Paths are compacted to reduce the total number of test patterns generated. In the case where test vectors have been generated for each path, the direct compaction of test vectors is called static compaction. Dynamic compaction is implemented in *CodGen* where the Necessary Assignments (NAs) of the paths are compacted to improve the compaction ratio. Dynamic compaction compares each of the successfully justified paths with all the previously found justified and compacted paths, stored in a pool. A successfully justified path is justified against each path in the pool until it can be compacted with one of them. So the time would increase in the order of justification time (which is increase in number of gates as explained in section 3.3) for all the comparisons (quadratic). Also for N paths, the time complexity is in the order of $O(N^2)$ as in the worst case a path is compared to all the previously found paths and results in no compaction. 2048x8 non mux has twice as many paths as 256x8 non mux (section 3.3). Hence, increase in cost would be of $2\cdot(3)^2 > 18$. However Table 5 shows that the CPU time for dynamic compaction from 256 non mux to 2048 non mux only increases by ~ 9. The compaction vectors comparison table on page 50 shows the number of vectors with 2 different pool sizes. We can see 2048 non mux has about 12 times the compaction vectors in 2048 mux which is the reason for increase in cost seen in Table 5.

Table 5. Number of Paths vs Compaction Time

Circuit	No. Sensitizable Paths	Dynamic Compaction Time(sec)
STC_1_256_8_mux	8854	2648.36
STC_1_256_8_non_mux	13623	7352.65
STC_1_2048_8_mux	10192	4331.78
STC_1_2048_8_non_mux	25208	66992.1

The above analysis assumes that a new test for a sensitizable path (a set of NAs) must be compared against all of the patterns (sets of NAs) in the path pool. In practice, the number of paths within each pattern in the pool starts out high and falls quickly. Most paths will fit within a pattern without comparing against too many patterns. And if a pattern has a large number of NAs, the comparison tends to quickly fail, since the pattern is "full". The success rate of dynamic compaction is low in all of the circuits. For example, in the largest circuit, the ratio of success to fail is 12473/194720. An additional factor is that to limit memory consumption, only a limited number of patterns are kept in an in-memory pattern pool for comparison. This pool is small compared to the total number of patterns. This combination reduces the compaction time.

3.5. All Factors Comparison of 256x8 Non-mux to 2048x8 Non-mux

We put together all of the factors computed above to determine the expected CPU time ratio between the 256x8 mux and 2048x8 non-mux circuits. The time goes as:

Total partial paths · (2 · Assignments on each path · length of each path) + Direct implications Cost

We have factor 2 for assignments in the above equation since we make the assignments, extend the path, and then destroy the assignments. Using the factors computed in the previous sections, the CPU time ratio between the two circuits is:

$$N1\cdot(2\cdot N2\cdot N4) + N3(\text{Section } 3.2.3) = 5\cdot(2\cdot3\cdot2.8) + 40 = 84 + 40 > 120$$

N1 = 5 Section 3.2.1, N2 = 3 Section 3.2.2, N4 = 2.8 Section 3.2.4.

This ratio is close to the ratio measured in Table 6. 31790/292 = 108.

Table 6. Number of Gates vs Path Generation Time

Circuit	Number of Gates	Path Generation Time(sec)
STC_1_256_8_mux	47777	197.411
STC_1_256_8_non_mux	56206	292.404
STC_1_2048_8_mux	92675	1169.16
STC_1_2048_8_non_mux	160246	31790.5

4. RESULTS

The circuits under consideration have the same length for the longest transition fault testable path. We have tried to change the partial path store to heap, and see the effect on SmartPERT heuristic on the circuit. We report the following results as follows.

4.1. Heap Implementation of Partial Path Pool

The partial path store is a priority queue currently. The growth of paths is always in the DFS style such that one of the paths in extended all the way and when it fails we retreat all the assignments and start with the path next in the store. However, if we have more insertions into the queue which is O(N), doing it in O(log N) would be more beneficial. To this, we have implemented the queue. However, for these circuits we see it is not beneficial. Figure 32 shows the highest Esperance partial path removed and all its fan outs inserted.

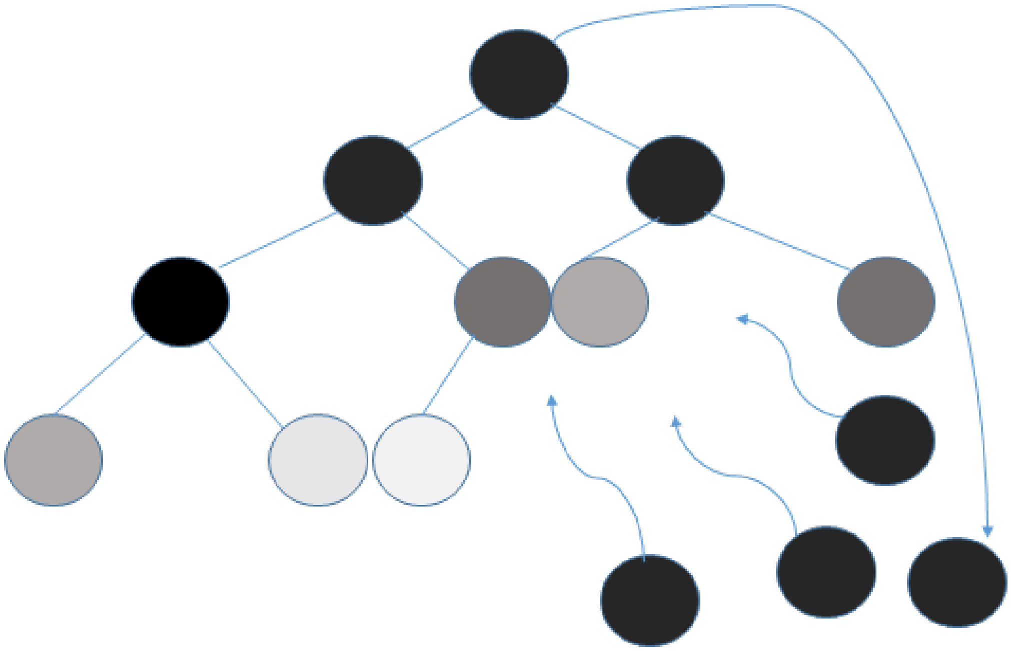


Figure 32. Max Heap on Esperance

The reason behind this could be for every pop out, the number of insertions vary a lot, let us say 20 fan outs and all of them needs to be inserted, a few of them would have Esperance as high as current ones, or most probably near the top of the heap now all of them would fight for the highest point since all the ones with equal Esperance have length greater than 1 than any other node in the heap….and the comparison that needed to be done are 10 fold in a heap of 1000… Whereas in a priority queue, the comparisons are limited since, insertion is from top. Again this may work better in some other circuits if the Esperance changes a lot with additional gates and fan outs of gates are very less. Table 7 compares the performance of heap to priority queue.

Table 7. Heap vs Priority Queue

Circuit	K	Total Time Taken DD:HH:MM:SS	Total Paths	Fault Coverage	Heap (size 500) Sorted on Esperance DD:HH:MM:SS	Total Paths	Fault Coverage
STC_1_256_8_mux	1	1:12:54	8854	0.36665	1:09:57	8854	0.36818
	2	2:30:09	13908	0.37171	2:30:49	13250	0.36013
STC_1_256_8_non_mux	1	3:16:09	13618	0.31994	3:58:50	13618	0.32077
	2	6:39:20	23441	0.32276	7:08:56	22742	0.316338
STC_1_2048_8_mux	1	3:09:42	10192	0.27664	2:55:21	9534	0.26929
	2	4:59:00	16180	0.27979	5:05:57	15579	0.27302
STC_1_2048_8_non_mux	1	1:10:35:32	25208	0.1941	2:00:50:42 (Heap 2000)	24532	0.192123
	2	2:04:42:54	46074	0.19505	2:10:34:38 (Heap 2000)	45358	0.19299

Table 8 shows the comparison between Heap sorted on Esperance and Heap sorted on Esperance and Length. Among the paths with same Esperance, the paths of higher length would be given preference. Circuit 2048*8 mux shows an improvement.

Table 8. Time Taken for Heap Sorted on Esperance vs Esperance and Length

Circuit	**Original Heap sorted on Esperance (size 500)** **DD:HH:MM:SS**	**Heap sorted on Esperance and Length(size 500)** **DD:HH:MM:SS**
STC_1_256_8_mux	1:12:54	1:08:38
STC_1_256_8_non_mux	3:16:09	3:21:59
STC_1_2048_8_mux	3:09:42	2:35:59
STC_1_2048_8_non_mux	1:10:35:32(Heap size 2000)	1:12:07:04(Heap size 2000)

4.2. SmartPERT Heuristic Enhancement to the Code

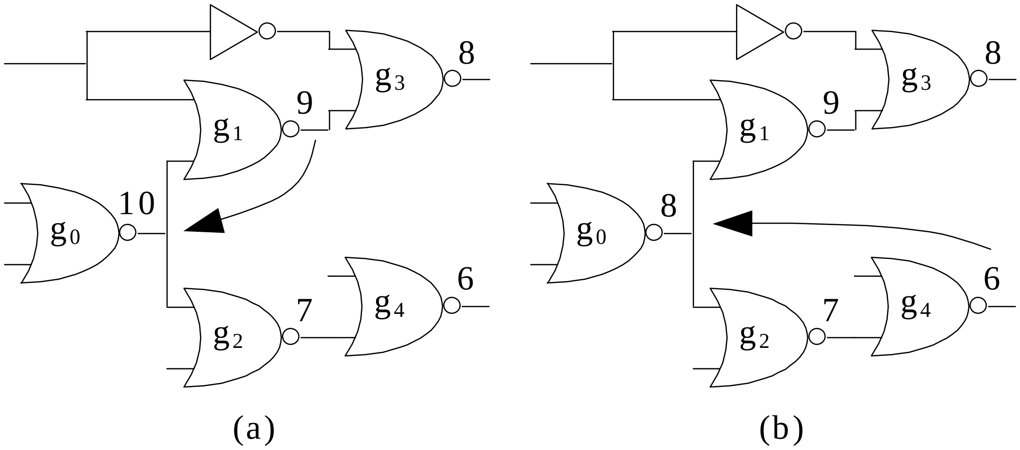


Figure 33. SmartPERT Heuristic Applied in Gates, reprinted with permission from [24]

The heuristic is as follows. In Figure 33(a), the PERT delay (Esperance) values are updated based on the immediate gates, but this is a blind comparison; most of the immediate conflicts are not updated here. Suppose S-PERT(g_i) is being computed. G = {g_j | g_j is d gates from g_i in g_i's fan-out tree}, and G is sorted by S-PERT(g_j) in decreasing order. The heuristic pops the first gate g_j in G and attempts to propagate a transition from g_i to g_j. If there is no conflict (the transition successfully reaches g_j, with all the constraints applied), S-PERT(g_i) is set to S-PERT(g_j) + d. Otherwise, it pops the second gate in G and repeats the same procedure. In Figure 33(b), for example, at first the heuristic tries to propagate a transition from g_0 to g_3, but finds it is impossible to set the side inputs of g_1 and g_3 both to non-controlling values. Then it tries g_4 and does not meet any conflict. So S-PERT(g_0) is 8.

It is obvious that increasing the S-PERT depth can make the S-PERT delays closer to the delay of the longest testable path from that gate to a primary output, but its cost increases exponentially. In this work the S-PERT depth is fixed at 2, but one option is to increment it if path searches repeatedly fail. The benefit of this heuristic depends heavily on the structure of the circuit. Table 9 shows the overall time taken comparison for all the circuits for K=1 and K=2.

Table 9. PERT vs SmartPERT

Circuit	K	Total running time DD::HH::MM::SS	Original Paths	Fault Coverage	Total Running Time with SmartPert DD::HH::MM:SS	Smart PERT paths	Fault coverage
STC_1_256_8_mux	1	1:12:54	8854	0.36665	1:10:23	8945	0.36845
	2	2:30:09	13908	0.37171	2:51:59	14008	0.37222
STC_1_256_8_ non_mux	1	3:16:09	13618	0.31994	3:50:18	13714	0.32095
	2	6:39:20	23441	0.32276	7:23:44	23541	0.32305
STC_1_2048_8_ mux	1	3:09:42	10192	0.27664	3:13:22	10350	0.27923
	2	4:59:00	16180	0.27979	5:32:23	16413	0.28175
STC_1_2048_8_ non_mux	1	1:10:35:32	25208	0.1941	2:06:38:55	25309	0.19441
	2	2:04:42:54	46074	0.19505	3:08:25:30	46191	0.19516

In Table 9, we can see that the total time taken for path generation for all the circuits has increased. We tried to find the root cause of this. Analyzing the time distribution for various modules of the algorithm, we observed an increase in factor N1 (total number of partial paths processed) described in section 3.2.1. This increases both the initialization and direct implications cost. We propose the following hypothesis for the increase in total number of partial paths. S-PERT only updates the Esperance based on conflicts at depth 2. Similar failures can occur at greater depths which are not considered. In Figure 33 let g_0 be one of the fan-outs of a gate g_i. Other fan-outs of g_i are more likely to have an Esperance of 10 than 8, from the gate level perspective. Since S-PERT has not updated their Esperance, they are more likely to grow to be false paths or decrease their Esperance much later in the process. When g_i is extended to all its fan outs, all of them will be of same length, but the ones with higher Esperance would be processed first. With S-PERT updating g_0, there is only a delay added before looking at the partial path formed from g_i to g_0. In case the circuit structure is such that all the other fan-out paths would extend to two or three additional levels before failing or decrease their Esperance to less than 8, S-PERT only increases the number of partial paths processed.

4.3. Other Heuristics

4.3.1. Dynamic Compaction Path Pool Size Variation

We have tried varying dynamic path pool size since there is significant amount of time going into this module. The benefit is obvious as seen in Table 10. We do not want to compare each path with every other path found. We restrict the number of paths to be compared with the path pool size. Also, if a path matches necessary assignments with another path, they are not necessarily compactable. Hence, an upper limit on the number of times it can fail compaction is defined. Table 10 shows the comparison between the time taken for pool sizes 2000 and 1000. The compactions failure limits are set at 1000 and 500 respectively. The number of compaction vectors increased for all the circuits is shown in Table 11.

Table 10. Dynamic Compaction Pool Size 2000 vs 1000

Circuit	Pool 2000 Compaction fails 1000 DD:HH:MM:SS	Paths	Pool 1000 Compaction fails 500 DD:HH:MM:SS	Paths
STC_1_256_8_mux	1:12:54	8854	1:09:19	8854
STC_1_256_8_non_mux	3:16:09	13618	3:04:00	13623
STC_1_2048_8_mux	3:09:42	10192	2:59:34	10192
STC_1_2048_8_non_mux	1:10:35:32	25208	1:07:33:56	25208

Table 11. Compaction Vectors with Dynamic Path Pool Size 2000 vs 1000

Circuit	No. of Vectors Pool size 2000 Compaction fails 1000	No. of Vectors Pool size 1000 Compaction fails 500
STC_1_256_8_mux	930	930
STC_1_256_8_non_mux	1603	1622
STC_1_2048_8_mux	946	946
STC_1_2048_8_non_mux	12735	13971

4.3.2. All Fan-out Extension

This is an attempt to save the cost of direct implications. The heuristic is implemented as a modification of the KLPG flow. When extending a gate to its fan outs, direct implications are computed to check conflicts before updating the Esperance and adding the path to the partial path pool. If the average number of fan outs is not too large, let us extend to all the fan outs, by just assigning a transition at each fan out gate and not check any conflicts. Once the path with highest Esperance is selected, it is checked for conflicts from direct implications on the frontier gate. This would improve the performance if the average fan out is not creating too many paths for the partial path pool (current size 500). However, we have seen too many partial paths created and have not pursued this heuristic completely. A modification of this heuristic is to keep extending the partial paths based on the maximum Esperance without performing any direct implications and let SAT work on justifying the paths. We performed this on the 256x8 mux and 2048x8 mux circuits. Tables 12 and 13 show the results of this.

Table 12. Total Time Taken

Circuit	Total Paths	Total Time Taken (HH::MM::SS)	Total Paths heuristic	Total Time Taken by heuristic (HH::MM::SS)
STC 256x8 mux	8854	1:12:54	7722	6:44:00
STC 2048x8 mux	10192	3:09:42	8415	14:19:26

Table 13. Total Paths Processed by SAT Engine and Time Taken

Circuit	Total Paths Success/Fail by SAT engine	Total Time for Justification (sec)	Total Paths Success/Fail by SAT engine using heuristic	Total Time for Justification (sec) with heuristic
STC 256x8 mux	12161/7213	1316.42	191891/23470	20697.7
STC 2048x8 mux	20489/12883	4384.92	205685/29852	46416.8

There is a huge increase in the total number of paths and the total number of false paths. The kind of paths failing and passing are not analyzed in this research. However, false paths can be used to identify the kind of logic failing in the original algorithm and further find the reason for this. The paths through memory should be passing for all the bits if a path is generated for one bit and so is the case with each byte. Unless the logic is unable to write to some of the memory locations, there should not be any failures in similar paths going to various bytes of the memory in the write path. In the read path, there is a unique path from each byte to the output. There should not be any failures in the paths as long as we are able to launch a transition on a line out of the memory as in the chance of conflicts here is rare. Most of our research has been agnostic of the circuit structure, but circuits need to be explored for analyzing failed paths.

5. CONCLUSION AND FUTURE WORK

In this work, we have developed a mathematical reasoning for the super linear increase in time taken for path generation for large circuits. We have implemented various heuristics and observed the response of circuits. An improvement was seen only in few cases while the tool showed the same or degraded performance to these heuristics. We believe for a substantial better performance an effective data structure to process each of the partial paths in the partial path store is needed.

Looking at the current implementation, a data structure with O(NlogN) or O(kN)/superlinear time, would bring substantial advantage. For instance, we have observed in most of the cases that the path extended is the path that has just been processed. If a partial path is extended for 5 gates before deviating from the course of the path, (taking a path from previously split Gate because of failure or larger Esperance), the reconstruction and destruction of all the assignments is done 5 times in an exhaustive way. It would be very beneficial if we can predict when a just processed path is extended. In case prediction is not possible or results in higher time complexity, we can keep the assignments as it is in the circuit until we decide the next path to be selected. Another way would be to implement a data structure that can hold a list of assignments made on partial paths and to find if the extended gate fits in this without conflicts. Instead of recursively doing direct implications many times on a gate, these can be pre computed and stored to check if they conflict with the current state of circuit in a plug and play fashion. One of the other reasonable ways to do this would be to store direct implications of a gate for a particular transition on one of its inputs. By far most of the logic gates used in the circuits have two inputs and one output; hence they would probably fit in a hash table based on gate ID and transition. The direct implications now for every path are run for each gate and repetition of the gates is not considered between various partial paths. A pre-processed hash table for all the gates with rising and falling transitions certainly would reduce CPU time. However, there would be a tradeoff between space and time. Yukun's work [46] includes the reason to avoid storing assignments based on each path.

In section 3.2.1, we explained how the total number of partial paths will increase the total time taken. However, the actual number can be known only after running a complete test on the circuit. We can pull periodic statistics from the tool on number of paths; however that would only explain the time elapsed. To estimate the time for any new circuit, a mathematical formula in terms of total gates or total fan out needs to be developed. We looked at the average fan out but that doesn't seem to explain the increased partial paths. This could be more a mathematical formula in terms of gate type and corresponding fan out to estimate the total number of partial paths it may produce. However, to capture it in a formula sufficient circuits should be tested and possibly find a pattern of gates vs fan outs.

The numbers of repetitive paths that are validated by the SAT engine are huge. When we are trying to update the coverage the gates in this path have already been covered by previously found paths. The total paths generated are 3 times the actual paths of interest. This is easy to avoid by just adding a hash engine for each of the paths generated and check if the path would update coverage, prior to sending it to the SAT engine.

There is a scope of optimization in the SAT engine as well. Especially in the case of larger circuits, more than half of the time is spent on justification. Since the SAT engine was not developed in our lab and had better performance compared to PODEM [46], it was plugged in to the tool. More SAT engines optimized to memory circuits could be used.

Dynamic compaction could be further analyzed to minimize the quadratic time component of checking each path against many patterns. A hash table of size 2000 instead of a pool of size 2000 can be implemented to compare two paths for assignments. There are number of calls to SAT engine from Dynamic Compaction module. Additional justification is done in the cases of pushing a path out of dynamic compaction path pool, if total paths compacted in a pattern are more than 5000, if a path failed to compact with any newly arrived paths for more than a certain limit, writing a vector to a file (this is only when corresponding flag is enabled). A few of these could be redundant.

The above ideas require changes to data structures and might change the flow in various modules. Some of them the modules are working on the partial path store as a list. Insertion and traversal in a list could change to addition and look up in a table. Overall data structural modification might also call for a development of an entirely new tool. The changes required are huge and introduce new bugs that might take significant research time to solve.

REFERENCES

[1] R. D. Eldred, “Test Routines Based on Symbolic Logical Statements”, Journal of the ACM, vol. 6, pp. 33-36, Jan 1959.

[2] J. A. Waicukauski, E. Lindbloom, B. K. Rosen, and V. S. Iyengar, “Transition Fault Simulation”, IEEE Design and Test of Computers, vol. 4, no. 2, pp. 32-38, Apr. 1987.

[3] K. T. Cheng, “Transition Fault Testing for Sequential Circuits”, IEEE Transactions on Computer Aided Design of Integrated Circuits and System, vol. 12, no.12, pp. 1971-1983, Dec. 1993.

[4] Z. Barzilai and B. K. Rosen, “Comparison of AC Self-Testing Procedures”, in Proceedings of IEEE International Test Conference (ITC), pp. 89-94, Oct. 1983.

[5] G. L. Smith, “Model for Delay Faults Based Upon Paths”, in Proceedings of IEEE International Test Conference (ITC), pp. 342-349, Oct. 1985.

[6] V. S. Iyengar, B. K. Rosen, and I. Spillinger, “Delay Test Generation 1 – Concepts and Coverage Metrics”, in Proceedings of IEEE International Test Conference (ITC), pp. 857-866, Sep. 1988.

[7] V. S. Iyengar, B. K. Rosen, and I. Spillinger, “Delay Test Generation 2 – Algebra and Algorithms”, in Proceedings of IEEE International Test Conference (ITC), pp. 867-876, Sep. 1988.

[8] J. Carter, V. Iyengar, and B. Rosen, “Efficient Test Coverage Determination for Delay Faults”, in Proceedings of IEEE International Test Conference (ITC), pp. 418-427, Sep. 1987.

[9] E. S. Park, M. R. Mercer, and T. W. Williams, “Statistical Delay Fault Coverage and Defect Level for Delay Faults”, in Proceedings of IEEE International Test Conference (ITC), pp. 492-499, Sep. 1988.

[10] C. W. Tseng and E. J. McCluskey, “Multiple-Output Propagation Transition Fault Test”, in Proceedings of IEEE International Test Conference (ITC), pp. 358-366, Oct. 2001.

[11] X. Lin and J. Rajski, “Propagation Delay Fault: A New Model to Test Delay Faults”, in Proceedings of IEEE Asian South Pacific Design Automation Conference, pp. 178-183, Jan. 2005.

[12] A. K. Pramanick and S. M. Reddy, “On the Fault Coverage of Gate Delay Fault Detecting Tests”, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 16, no. 1, pp. 78-94, Jan. 1997.

[13] A. K. Majhi, J. Jacob, L. M. Patnaik, and V. D. Agrawal, “On Test Coverage of Path Delay Faults”, in Proceedings of International Conference on VLSI Design, pp. 418-421, Jan. 1996.

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

[15] L. H. Goldstein and E. L. Thigpen, “SCOAP: Sandia Controllability/Observability Analysis Program”, in Proceedings of ACM/IEEE Design Automation Conference, pp. 190–196, Jun. 1980.

[16] E. B. Eichelberger and T. W. Williams, “A Logic Design Structure for LSI Testability”, in Proceedings of ACM/IEEE Design Automation Conference, pp. 462-468, Jun. 1977.

[17] F. Motika, N. Tendolkar, C. Beh, W. Heller, C.Radke et al., “A Logic Chip Delay-test Method Based on System Timing”, IBM Journal of Research and Development, Vol. 34, No.2/3, pp. 299-312, March/May 1990.

[18] S. DasGupta, P. Goel, R. G. Walther and T. W. Williams, “A Variation of LSSD and Its Implications on Design and Test Pattern Generation in VLSI”, in Proceedings of IEEE International Test Conference (ITC), pp. 63-66, Nov. 1982.

[19] C. T. Glover and M. R. Mercer, “A Method of Delay Fault Test Generation”, in Proceedings of ACM/IEEE Design Automation Conference, pp. 90-95, Jun. 1988.

[20] B. I. Dervisoglu and G. E. Strong, “Design for Testability: Using Scan Path Techniques for Path-delay Test and Measurement”, in Proceedings of IEEE International Test Conference (ITC), pp. 365-374, Oct. 1991.

[21] J. Savir, “Skewed-Load Transition Test: Part I, Calculus”, in Proceedings of IEEE International Test Conference (ITC), pp. 705-713, Sep. 1992.

[22] S. Patel and J. Savir, “Skewed-Load Transition Test: Part II, Coverage”, in Proceedings of IEEE International Test Conference (ITC), pp. 714-722, Sep. 1992.

[23] J. Savir and S. Patel, “Broad-Side Delay Test”, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 13, no. 8, pp. 1057-1064, Aug. 1994.

[24] W. Qiu and D. M. H. Walker, “An Efficient Algorithm for Finding the K Longest Testable Paths Through Each Gate in a Combinational Circuit”, in Proceedings of

IEEE International Test Conference (ITC), pp. 592-601, Sep. 2003.

[25] W. Qiu, J. Wang, D. M. H. Walker, D. Reddy, X. Lu et al., "K Longest Paths Per Gate (KLPG) Test Generation for Scan-Based Sequential Circuits", in Proceedings of IEEE International Test Conference (ITC), pp.223-231, Oct. 2004.

[26] D. M. H. Walker, "Tolerance of Delay Faults", in Proceedings of IEEE International Workshop on Defect and Fault Tolerance in VLSI Systems, pp. 207-216, Nov. 1992.

[27] P. Pant and J. Zelman, "Understanding Power Supply Droop during At-Speed Scan Testing", IEEE VLSI Test Symposium, pp.227-232, May 2009.

[28] B. Nadearu-Dostie, K. Takeshita and J. F. Cote, "Power-Aware At-Speed Scan Test Methodology for Circuits with Synchronous Clocks", in Proceedings of IEEE International Test Conference (ITC), pp.1-10, Oct. 2008.

[29] D. Belete, A. Razdan, W. Schwarz, R. Raina, C. Hawkins et al., "Use of DFT Techniques in Speed Grading a 1 GHz+ Microprocessor", in Proceedings of International Test Conference (ITC), pp. 1111-1119, Jan. 2002.

[30] J. Zeng, M. Abadir, G. Vandlin, L. Wang, A. Kolhatkar et al., "On Correlating Structural Tests with Functional Tests for Speed Binning of High Performance Design", in Proceedings of IEEE International Test Conference (ITC), pp. 31-37, Oct. 2004.

[31] M. Abadir, S. Magdy, and H. K. Reghbati, "Functional Testing of Semiconductor Random Access Memories", ACM Computing Surveys 15.3, pp. 175-198, 1983.

[32] K. Zarrineh, S. J. Upadhyaya, and S. Chakravarty, “A New Framework for Generating Optimal March Tests for Memory Arrays”, in Proceedings of IEEE International Test Conference (ITC), pp. 73-82, Jan. 1998.

[33] R. Rajsuman, “Design and Test of Large Embedded Memories: An Overview”, Design Test of Computers, IEEE, vol. 18, no. 3, pp. 16-27, May 2001.

[34] K. Zarrineh and S. J. Upadhyaya, “On Programmable Memory Built-in Self Test Architectures”, in Proceedings of Design, Automation and Test in Europe Conference and Exhibition (DATE), pp. 708-713, Mar. 1999.

[35] A. L. Crouch, J. L. McKeown and C. G. Shepard, “Multiple BIST Controllers for Testing Multiple Embedded Memory Arrays”, U.S. Patent 5,995,731., 30 Nov. 1999.

[36] M. Tuna, M. Benabdenbi and A. Greiner , “At-Speed Testing of Core-Based System-on-Chip Using an Embedded Micro-Tester”, IEEE VLSI Test Symposium, pp. 447-454, May 2007.

[37] D. E. Ross, T. Wood and G. Giles, “Conversion of Small Functional Test Sets of Nonscan Blocks to Scan Patterns”, in Proceedings of IEEE International Test Conference (ITC), pp. 691-700, Oct. 2000.

[38] F. Yang and S. Chakravarty, “Testing of Latch Based Embedded Arrays Using Scan Tests”, in Proceedings of IEEE International Test Conference (ITC), pp. 1-10, Nov. 2010.

[39] M. Banga, N. Rahagude and M. S. Hsiao , “Design-for-Test Methodology for Non-Scan At-speed Testing”, in Proceedings of Design, Automation Test in Europe Conference Exhibition (DATE), pp. 1-6, Feb. 2011.

[40] S. Lahiri, D. M. H. Walker and K. Bian, “KLPG based Pseudo-functional Test with Dynamic Compaction”, SRC TECHCON, Austin, TX, Sep. 2011.

[41] Z. Wang and D. M. H. Walker, “Dynamic Compaction for High Quality Delay Test”, IEEE VLSI Test Symposium, pp.243-248, Apr-May 2008.

[42] S. K. Goel and K. Chakrabarty, “Testing for Small-Delay Defects in Nanoscale CMOS Integrated Circuits”, Ch.2, Duncan M. Walker, 2013.

[43] N. Eén and N. Sörensson, “An Extensible SAT solver”, in Proceedings of International Conference on Theory and Applications of Satisfiability Testing, pp. 502-518, May 2004.

[44] R. Drechsler, S. Eggersglüβ, G. Fey and D. Tille “Test Pattern Generation using Boolean Proof Engines”, Ch. 10, Springer, Berlin, Germany, 2009.

[45] R. Drechsler, S. Eggersglüβ, G. Fey, A. Glowatz, F. Hapke et al., “On Acceleration of SAT-based ATPG for Industrial Designs”, IEEE Transactions on Computer-Aided Design, vol. 27, no. 7. pp. 1329-1333, Jul. 2008.

[46] Y. Gao, T. Zhang, D. M. Walker, “Pseudo Functional Path Delay Test through Embedded Memories”, Journal of Electronic Testing: Theory and Applications, Volume 31 Issue 1, Pages 35-42, Feb. 2015.

[47] M. Sharma and J. H. Patel, “Finding a Small Set of Longest Testable Paths that Cover Every Gate”, in Proceedings of IEEE International Test Conference (ITC), pp. 974-982, Oct. 2002.

[48] J. A. Bell, “Timing Analysis of Logic-Level Digital Circuits Using Uncertainty Intervals”, M. S. Thesis, Department of Computer Science, Texas A&M University, Aug. 1996.

[49] L. T. Wang, C. E. Stroud and N. A. Touba, “System-on-Chip Test Architectures Nanometer Design for Testability”, Ch.2, Ch. 6, Morgan Kaufmann 2010.

[50] J. M. Rabaey, “Digital Integrated Circuits: A Design Perspective”, Prentice-Hall, 1996.