PSEUDOFUNCTIONAL DELAY TESTS FOR HIGH QUALITY

SMALL DELAY DEFECT TESTING

A Thesis

by

SHAYAK LAHIRI

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

MASTER OF SCIENCE

December 2011

Major Subject: Computer Engineering

Pseudofunctional Delay Tests for High Quality Small Delay Defect Testing

PSEUDOFUNCTIONAL DELAY TESTS FOR HIGH QUALITY

SMALL DELAY DEFECT TESTING

A Thesis

by

SHAYAK LAHIRI

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

MASTER OF SCIENCE

Approved by:

Chair of Committee,   Duncan Walker
Committee Members,   Jyh-Charn Liu
                       Sunil Khatri
Head of Department,   Duncan Walker

December 2011

Major Subject: Computer Engineering

ABSTRACT

Pseudofunctional Delay Tests for High Quality Small Delay Defect Testing.

(December 2011)

Shayak Lahiri, B.Tech., Motilal Nehru National Institute of Technology, Allahabad,

India

Chair of Advisory Committee: Dr. Duncan Walker

Testing integrated circuits to verify their operating frequency, known as delay testing, is essential to achieve acceptable product quality. The high cost of functional testing has driven the industry to automatically-generated structural tests, applied by low-cost testers taking advantage of design-for-test (DFT) circuitry on the chip. Traditional at-speed functional testing of digital circuits is increasingly challenged by new defect types and the high cost of functional test development. This research addressed the problems of accurate delay testing in DSM circuits by targeting resistive open and short circuits, while taking into account manufacturing process variation, power dissipation and power supply noise. In this work, we developed a class of structural delay tests in which we extended traditional launch-on-capture delay testing to additional launch and capture cycles. We call these Pseudofunctional Tests (PFT). A test pattern is scanned into the circuit, and then multiple functional clock cycles are applied to it with at-speed launch and capture for the last two cycles. The circuit switching activity over an extended period allows the off-chip power supply noise transient to die down prior to the at-speed

launch and capture, achieving better timing correlation with the functional mode of operation. In addition, we also proposed advanced compaction methodologies to compact the generated test patterns into a smaller test set in order to reduce the test application time. We modified our *CodGen* K longest paths per gate automatic test pattern generator to implement PFT pattern generation. Experimental results show that PFT test generation is practical in terms of test generation time.

DEDICATION

To my parents

ACKNOWLEDGEMENTS

TABLE OF CONTENTS

LIST OF FIGURES

Page

Page

1 INTRODUCTION

Testing integrated circuits to verify their operating frequency, known as delay testing, is essential to achieve acceptable product quality. Delay test must not accept slow chips nor reject fast ones. The high cost of functional test has driven the industry to automatically-generated structural tests, applied by low-cost testers taking advantage of design-for-test (DFT) circuitry on the chip, such as scan chains, on-chip test pattern compression/decompression, built-in self-test (BIST) circuits, and test access mechanisms (TAM). The principal challenge in structural delay test is to achieve the same quality as functional and system test while minimizing the cost (test data volume, test application time, DFT circuit overhead) while minimizing overkill (rejecting good chips as bad). The delay test challenge is more difficult for chips fabricated in deep submicron (DSM) semiconductor technology. DSM technology has increased circuit delay variability, increased signal crosstalk, power supply noise, and requires complex circuit techniques to reduce power dissipation. This research addresses the problems of accurate delay test in DSM circuits by targeting resistive open and short circuits while taking into account manufacturing process variation, power dissipation and power supply noise.

The main goal of delay test is to detect timing defects in order to guarantee that the design meets the timing specifications. It has been observed that tests that do not specifically look for timing failures have a very low fortuitous detection rate.

_____

This thesis follows the style of *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.*

As the semiconductor industry pushes toward more aggressive timing requirements, the detection of small delay defects is becoming increasingly critical. One of the main aspects of testing is that it should be able to detect problems that may arise in the functional operation of the chip. This requires careful characterization of the functional mode and increased effort to get as close to it as possible to this environment while in test mode. Some important factors that need to closely correspond to functional mode are the power supply noise, signal coupling and power dissipation. This is critical in systems-on-chip (SOCs) and die stacks, where modeling the overall noise environment is very difficult, due to the unknown combinations of subsystems. Thus a test performed under different conditions may not be able to achieve the desired objectives. Tests performed under different operating conditions can lead to over testing and under testing, affecting yield, reliability, reputation and profits.

In light of the above discussion, it is clear that one of the most important structural test requirements is a high correlation to functional and system test. This research focuses on development of new structural test methods to target small delay defects with high functional and system test correlation in the presence of deep submicron effects.

2 MOTIVATION

The International Technology Roadmap for Semiconductors (ITRS) projects that design-for-test (DFT) methods must continue to improve to maintain product quality, while reducing the test cost per transistor [1]. Traditional at-speed functional testing of digital circuits is increasingly challenged by new defect types and the high cost of functional test development. Even when available, extensive functional test patterns still miss defects caught by structural tests. Functional tests also make it more difficult to diagnose defects for yield improvement. As noted in the roadmap, increasingly the focus of delay test is on screening small delay defects (SDD). It also notes that any new delay test method must minimize overkill by avoiding test of false paths [2] [3], and must minimize test data volume with aggressive test pattern compression [4].

One of the challenges of scan-based structural delay test is that it uses a slow-fast-slow clocking approach, with slow scan and fast functional clock cycles. In launch-on-capture (LOC) test, the time delay between the last scan-in cycle and the first functional cycle must be long enough to allow the scan enable (SE) signal to change. The scan cycle time is typically an order of magnitude slower than the functional cycle time, in order to minimize the area and power cost of scan chain routing and buffering. Since the power grid is designed for functional operation, it largely reaches its quiescent state prior to the first functional (launch) cycle. The power grid time constant due to off-chip inductance is much longer than the functional clock cycle, so it can take dozens of functional clocks before the inductor currents ramp up to supply the on-chip switching activity. In the meantime, this current must be supplied from on-chip parasitic and

decoupling capacitance, causing the supply voltage to droop. This droop is known as the *dI/dt effect*, and is the dominant power supply noise problem. This initial voltage droop causes the circuit to operate more slowly than in normal functional operation [5] [6]. However, the ringing of the supply can also cause the circuit to operate faster than in normal operation in some situations [5]. Essentially, there is a mismatch between scan and functional test speeds due to the mismatch between the functional and test power supply voltages.

Prior work [5] [6] has shown that one solution to the dI/dt problem is to apply a series of scan or functional cycles that are slower than functional speed, but much faster than scan speed, to ramp inductor currents prior to the launch and capture of the delay test. In this work, we extend traditional launch-on-capture delay testing to additional launch and capture cycles, which we term Pseudofunctional test (PFT). Our motivation is to be able to achieve a higher correlation to functional mode than traditional LOC test. In addition, we also propose advanced compaction methodologies to compact the generated test patterns into a reduced test set, in order to reduce the test application time. The merit of the proposed research is that it has the potential to significantly improve the quality of structural delay test, with the goal of being competitive with the accuracy of functional delay test at a much lower cost. The models, algorithms and software tools developed in this research can be used to help drive related research areas, such as test pattern compression and delay defect diagnosis.

3 PSEUDOFUNCTIONAL TEST

Pseudofunctional Test (PFT) is a class of structural delay tests, in which traditional launch-on-capture delay testing is extended to additional launch and capture cycles. A test pattern is scanned into the circuit, and then multiple functional clock cycles are applied to it, with at-speed launch and capture for the last two cycles. The circuit switching activity over an extended period allows the off-chip power supply noise transient to die down prior to the at-speed launch and capture. This helps us in reaching an operating environment close to functional mode when the launch and capture take place. This increases the delay test correlation between structural and functional models, and minimizes over and under testing.

The term Pseudofunctional test has previously been used in digital circuit testing to mean constraining test patterns to use functional or near-functional states [7] [8] [9] [10] [11] [12] [13] . In this work, we would develop Pseudofunctional structural delay tests in which a test pattern is scanned into the circuit, and then multiple functional clock cycles are applied to it, as shown in Fig. 1. If the initial state is functional, then these tests are short bursts of functional tests. We do not make an explicit effort to constrain the pattern generation to use functional states. The major focus of this work is to ensure that the circuit is close to functional operating mode when the test is being applied. If the initial pattern is a functional pattern, then we have the added advantage of applying functional patterns to the circuit under test in a functional or near functional mode. The nature of most circuit designs is that the starting state for a PFT test will be functional or near-functional.
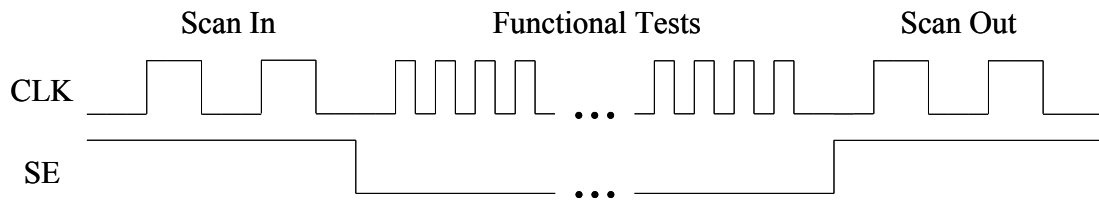
Fig. 1. Clocking of Pseudofunctional Test.

A Pseudofunctional test has multiple advantages. First, the power supply noise, signal coupling and power dissipation are similar to functional operation. This is critical in SOCs and die stacks, where modeling the overall noise environment is very difficult due to the unknown combinations of subsystems. Multi-cycle tests also permit test of time-borrowing between latches, and detection of metastability events triggered by setup or hold time violations. PFTs also permit detection of defects in power gating circuitry, and test of circuits using clock networks that require multiple cycles to stabilize, such as tuned LC networks.

PFT can also be used to address the dI/dt effect described in Chapter 2. One solution to the dI/dt problem is to apply a series of scan or functional cycles that are slower than functional speed, but much faster than scan speed, to ramp inductor currents prior to the launch and capture of the delay test [5] [6]. These cycles are termed *preamble cycles*, as shown in Fig. 2.

3.1 Preamble Cycles for PFT

Preamble cycles are a series of scan or functional cycles that are slower than functional speed, but much faster than scan speed. They ramp inductor currents prior to

the launch and capture of the delay test. The preamble must have the following characteristics:

1. Preamble cycles should be significantly longer than the chip-to-package time constant, so that transients have died down by the end of the preamble cycles.

2. The clock cycle time should be significantly shorter than the off-chip time constant, so that the off-chip supply can reach steady-state.

3. The preamble should propagate launch transitions on short paths to minimize the chances of test invalidation due to a delay fault caused by supply noise during preamble cycles.

4. There should be as few preamble cycles as possible in order to minimize automatic test pattern generation (ATPG) effort.

The chip-to-package time constant can be 30 or more functional cycles. The preamble clock cycles can be several times slower than functional clock cycles, so correspondingly fewer of them are required, meeting requirements 1, 2 and 4 [5] [6]. Requirement 3 can be met by using least-effort justification, which tends to find the shortest propagation paths. Fig. 2 shows a constant preamble cycle time, but it can be ramped down towards the functional cycle time, to minimize the voltage transient.
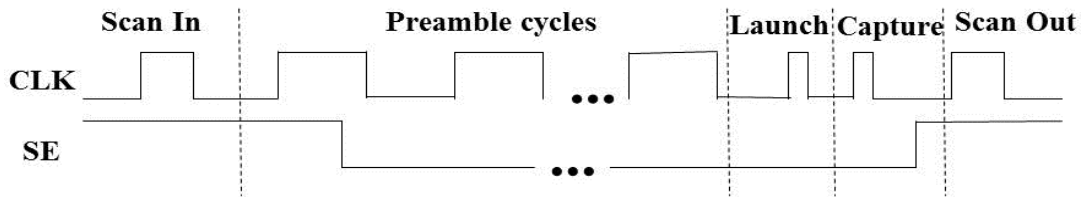
Fig. 2. Preamble Cycles Followed by At-Speed Launch and Capture Cycles in PFT.

The delay test that is launched and captured at speed must be justified back through the preamble cycles to the test pattern. From the viewpoint of the ATPG, these preamble cycles cause the circuit to appear as time frame expanded. This concept is illustrated in Fig. 3. Time frame expanded means that the vector in the nth time frame is derived from the circuit logic in the n-1th time frame. This can be also expressed as $V_n = C(V_{n-1})$. Here, V_n is the pattern of the nth time frame and V_{n-1} is the pattern of the n-1th time frame. As seen from the figure, a pattern can be initially shifted into the scan cells and the circuit response to it is captured back into the scan cells. This response is now applied back to the circuit again. We can alternatively view this as the concatenation of copies of the entire circuit including the scan cells as many times as there are clock cycles. Each of these copies corresponds to a single time frame.

It becomes the work of the ATPG to determine if the pattern in any particular time frame is going to cause an illegal state in any future time frame. Also, the ATPG needs to determine if there are any conflicts between the logical states of each gate in each of the time frames. It can be seen that with the increase in the number of time frames, the ATPG effort will increase significantly. Thus, one of the basic requirements of using PFT for delay test is to use a delay test model that is efficient and scalable.
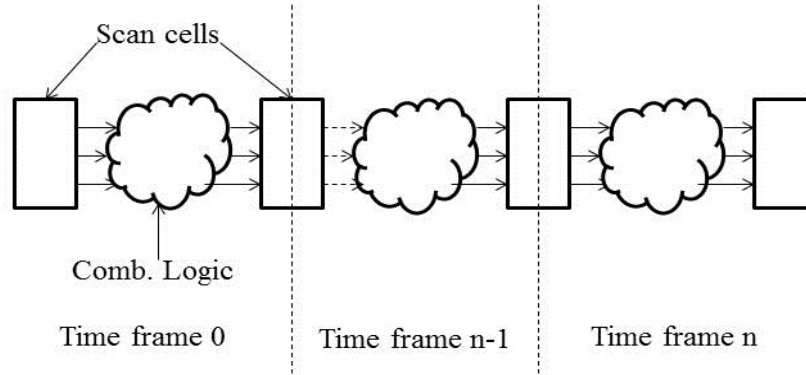
Fig. 3. Time Frame Expansion of the Circuit for PFT.

3.2 Extending LOC Path Delay Test for PFT

In launch-on-capture (LOC) test, the time delay between the last scan-in cycle and the first functional cycle must be long enough to allow the scan enable (SE) signal to change. The scan cycle time is typically an order of magnitude slower than the functional cycle time. Since the power grid is designed for functional operation, it largely reaches its quiescent state prior to the first functional (launch) cycle. The power grid time constant due to off-chip inductance is much longer than the functional clock cycle, so it takes dozens of functional clocks before the inductor currents can ramp up to supply the on-chip switching activity. In the meantime, this current must be supplied from on-chip parasitic and decoupling capacitances, causing the supply voltage to droop. This causes a poor correlation between test and functional operating conditions.

In order to avoid this, the traditional LOC model [14] must be modified to support preamble cycles. For the purposes of path delay test, we set a transition at the output of every gate of the target path. In the traditional LOC model, the transition at the start of

the path is caused by the value at the data input of the scan cell being opposite of the scanned in value, e.g. if the scanned in value is a 0 and the data input is a 1, then clocking the cell will cause a rising transition at the cell output. The ATPG problem is to go back one time frame to determine the initial test pattern that will generate the required transition at the start of the path and the necessary assignments to propagate transitions along the path to a capture point. With the introduction of preamble cycles, this model must be revised. We still want to justify the transitions for detecting the delay faults along the path in the last two time frames. However, we need to find the pattern which would cause these transitions from a time frame that is n cycles back, where n is the number of preamble cycles that we are applying. The traditional model for LOC described the initial and the final states of the circuit. Transitions were set based on values assigned in these states. With preamble cycles, these two states are now the 2nd to last and the last time frame. In traditional LOC, the test pattern was scanned in on the initial cycle. Now, the pattern must be scanned in on the 0th cycle. This is illustrated in Fig. 4. In other words, the final state of the circuit is moved to the last time frame. The initial state is moved to the 2nd to last time frame for setting the transitions, while the pattern application state is now the 0th cycle. The ATPG tries to find a pattern, which when applied in the 0th time frame, will cause the required transitions to occur going from the 2nd to last to the last time frame.
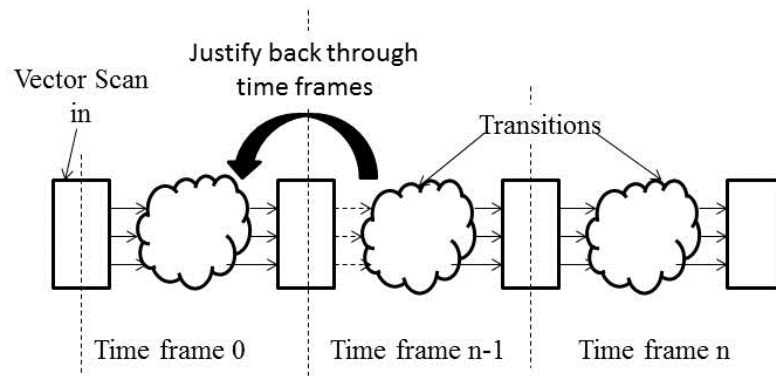
Fig. 4. Extending the Traditional LOC Model for Preamble Cycles in PFT.

4 KLPG FOR PSEUDOFUNCTIONAL TEST

The KLPG algorithm generates the K longest rising and falling paths through a target line under robustness constraints. The search space for each fault site, as shown in Fig. 5, is the fan-in and fan-out paths of the target line. In this work, we will combine KLPG path generation with PFT. In other words, we propose to generate KLPG patterns based on the PFT model. CodGen is a path delay test ATPG engine to generate the K Longest Paths per Gate (KLPG) [15] [16]. In order to make CodGen able to generate patterns for PFT, it will be modified to extend its capabilities beyond traditional LOC test. We will incorporate the model that we have described in Section 3.2. This makes the problem of path generation and path justification much harder than before. Since we are no longer following the traditional LOC test, the term initial and final clock cycles do not convey any meaningful information. Instead we use the term 'time frame' to designate each clock cycle from the scan in of the pattern to the capture cycle. The pattern is scanned in at time frame 0 and for a 16 cycle test; the capture is in time frame 15. The state of the circuit in any time frame is derived from its previous time frame.
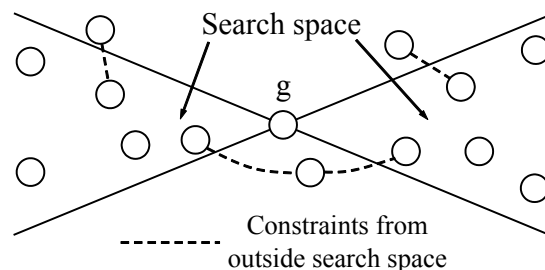
Fig. 5. KLPG Path Search Space.

There are three major steps in the KLPG process:

1. Path Initialization: The set up of the Launch Points in the circuit. A launch point can be a Primary Input (PI) or a Pseudo Primary Input (PPI).

2. Path Growth: Extending the path by adding one gate at a time, so that it extends from a Launch Point to a Capture Point. A Capture Point can be a Primary Output (PO) or Pseudo Primary Output (PPO).

3. Final Justification: Finding the test pattern as well as checking compatibility between all internal assigned values in the circuit.

We describe each of these processes in greater detail in this section. Due to the incorporation of the PFT model in the ATPG process, the Path Generation and Final Justification algorithms needed to be modified. A high level flowchart for KLPG is shown in Fig. 6. The algorithm is described in Fig. 7.
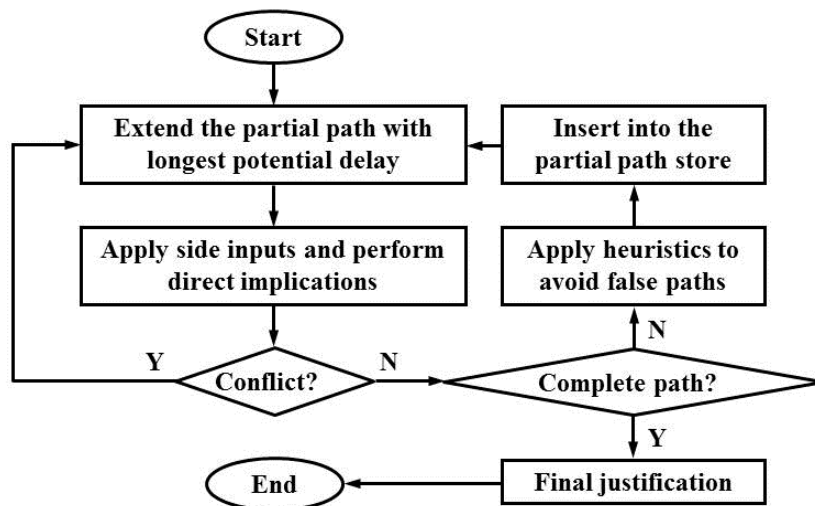
Fig. 6. Overview of KLPG Path Generation.

1. Parse the input files and perform pre-processing steps.

2. **For** each target fault site, until K paths has been generated or no more possible

3. Initialize the paths from the target fan-in cone Launch Points.

4. Add these to the Partial Path Store.

5. Extract the partial path with maximum Esperance.

6. Extend the extracted path.

7. Add side input constraints and perform Multi-Frame Direct Implications.

8. **If** a conflict is detected

9. Trash this partial path

10. **End If**

11. **Else If** Complete path formed

12. Perform Final Justification.

13. **End If**

14. **Else If** Complete Path is not formed

15. Apply false path elimination heuristics

16. Update the Esperance.

17. Re-insert in a sorted fashion into the Partial Path Store

18. Go to step 6.

19. **End If**

20. **End For**

Fig. 7. KLPG Algorithm.

4.1 Path Initialization and Growth

KLPG requires us to generate K longest rising and falling paths through a target line under robustness constraints. In order to activate the small delay defects across any of these long paths, we set transitions along the path, going from the 2nd to last to the last time frame. Transitions are set along each gate of the path. A transition can be of two types: rising or falling. A rising transition is one where the value on the 2nd to last time frame at the output of a gate is 0 and it changes to 1 in the next time frame. A falling transition is one where the value changes from 1 to 0. The path generation procedure has been derived from [15] and extended so that it is able to generate paths for the PFT model. When the number of preamble cycles are reduced to zero, the Launch on Capture model and PFT model become the same.

We perform preprocessing to help in the path generation process. We compute the maximum delay of each gate to the capture points. This is done without taking logic constraints into account. This is known as PERT or STA analysis and these delay values are known as PERT or STA delays. We also compute the SCOAP [17] values of the gates in the circuit. SCOAP values are used to guide the test generation and justification process.

In the initialization phase, we first identify the target fan-in cones and the target fan-out cones for the activation and propagation of a fault at any particular fault site. As shown in Fig. 5, this is the fan-in and the fan-out cone of that particular fault site. Path initialization starts with assigning transitions going from the 2nd last to the last time frame at every launch point in the target fan-in cone. All of these are considered partial

paths. A partial path is a path that has originated from a launch point but has not reached a capture point [14]. Partial paths are associated with an *Esperance* value, which is the sum of the PERT delay of the last node on the partial path and the total delay of the partial path. This Esperance serves as the upper bound of delay for any path, from a launch point to a capture point, and containing the partial path. Path growth takes place by adding a gate to the partial path with the highest Esperance and setting a transition on it going from the 2nd last to the last time frame. In the case of adding a multiple fan-out gate, we split the partial path. Based on the robustness criterion (e.g. robust, non-robust, long transition) we assign the constraints to propagate the transition. In order to quickly propagate the constraints through the circuit, we need to perform Direct Implication [15] over each of the time frames. This process continues iteratively unless we encounter conflicts and drop the partial path, or we reach a capture point.

When a partial path has not reached a capture point, false path elimination techniques are applied to trim off any false paths [15]. Then the path is re-inserted into the partial path store with an updated Esperance. Once a complete path is achieved, a Final Justification procedure is invoked. This checks the compatibility of indirectly implied logical values and finds a test pattern for the path. In the next sections, we describe the Direct Implication and the Final Justification procedures.

4.2 Multi-Frame Direct Implication

Previous work has shown that immediate identification of uniquely determined signals helps in speeding up the search process [18] [19]. We use the same philosophy during path generation and justification. During the path generation process, we set the

necessary transitions in the last two time frames. Each of these assignments, and the combination of previously discovered assignments, imply other logic values in the circuit. In order to determine these logic values, we use a process that we term Multi-Frame Direct Implication. It uses the basic concepts of Direct Implication [15] but it is able to discover and set implications across multiple time frames. Another potential benefit of this approach is that it is able to detect conflicts in assignments much earlier and trim off false paths at an earlier stage. This speeds up the search process. Multi-Frame Direct Implication is also able to propagate through Scan Cells into previous time frames, to yield maximum benefits.

Multi-frame Direct Implication is based on the following principle: the output value of a gate in time frame *j* is derived from the inputs to the gate in time frame *j* or vice versa. Multi-frame Direct Implication is performed on scanned flip-flops as well. The output of the scan cell in the time frame *j* is propagated backwards into time frame *j-1* on its data input line (and thus on the fan-in cone of the input line in time frame *j-1*). This is a bidirectional implication, which means that a value on the input of the scan cell is propagated forward into the next time frame through the data output line (and thus on the fan-out cone of the output of the scan call into the circuit).

An example of Multi-Frame Direct Implication is shown in Fig. 8. In the scan cell, the 1 at the input of the scan cell is derived from the output of the scan cell in the previous time frame. The 0 at the output of the scan cell is derived from the input of the scan cell in the next time frame. For the Backward implication through the AND gate, initially the values of 1, 0 and 1 are set at the output of the gate in three different time

frames. Let us consider that the inputs do not have any values on them, i.e. they are all X. The output value 1 implies input values 1 in the same time frame on both the input lines. This is an example of a uniquely determined signal. However, the output value of 0 cannot imply any value on the input lines. This is because a value of 0 on any input line (or both) will cause this output value. So the input values cannot be uniquely determined in this case, and thus remain X. Similarly, in the case of forward implication, let us assume we set 0, X and 1 on one input and 1, 0 and X on the other input in three different time frames. Also, let us assume that the outputs were all initially X. In the 1st time frame, the combination of 0 and 1 gives the signal 0 at the output. In the 2nd time frame, the combination of X and 0 gives us the value 0. However in the 3rd time frame, 1 and X give no uniquely determined signal and the output remains X.

One important aspect of Multi-Frame Direct Implication on logic gates is that it treats the values on the same gate at different time frames as independent logic values. This is correct because values on the same gate in a different time frame do not directly affect each other. The value may be propagated backwards or forwards into a scan cell, move into a new time frame and then trace back along a path to eventually reach the same gate again in a different time frame. However, in this case, the inputs or outputs of the particular gate already have the implied values and thus we need not worry about a direct relation between different time frame values on the same logic gate.
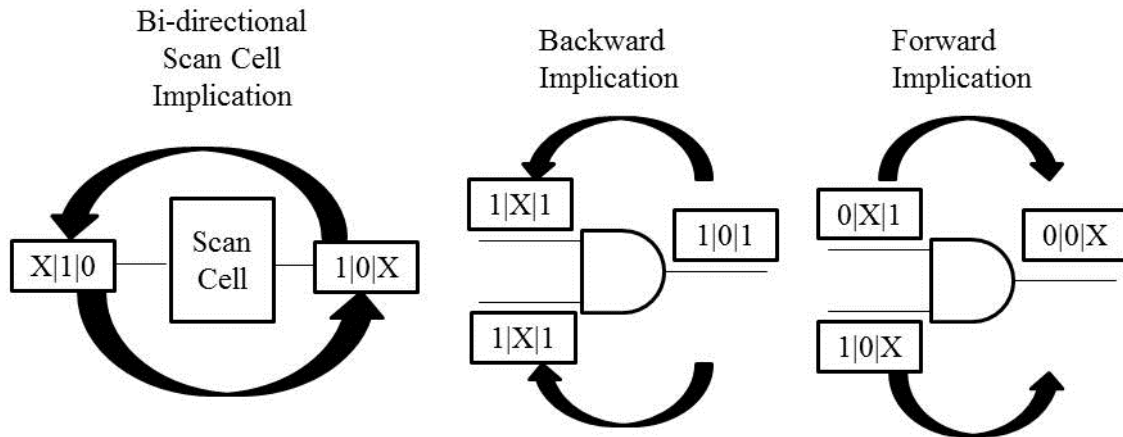
Fig. 8. Multi-frame Direct Implications.

It is to be noted that when we are not able to uniquely determine the values at the inputs or outputs of a gate by implications, we still need to ensure that these values do not cause a conflict. This is determined by Final Justification. A gate, is termed an Unjustified Gate [19] when an input is not uniquely determined by the value on the output line (and input lines that have a non-X value). Final justification ensures that each unjustified gate is justified and also helps in finding a vector for the path.

4.3 Final Justification for PFT

When a complete path is found, we need to perform Final Justification. The two major goals of this process are to justify each unjustified gate, and finding a pattern for testing the path that has been generated. An unjustified gate is one which has a non-X value, i.e. 0 or 1 on its output in a particular time frame; however the logical operation on its inputs in the same time frame yields an X. This has already been discussed in Section 4.2. A gate can be unjustified in multiple time frames. This means that we need

to justify the logic values at the gate output for each time frame. As discussed in Section 4.2 we treat the outputs of the same gate in different time frames as independent problems. The Final Justification procedure justifies values back to the 0th time frame, since that is the only time frame where we can set assignments by scanning in a test pattern. We have built on the PODEM [20] based procedure for LOC test, present in CodGen. The high level flow is shown in Fig. 9 and described in Fig. 10.
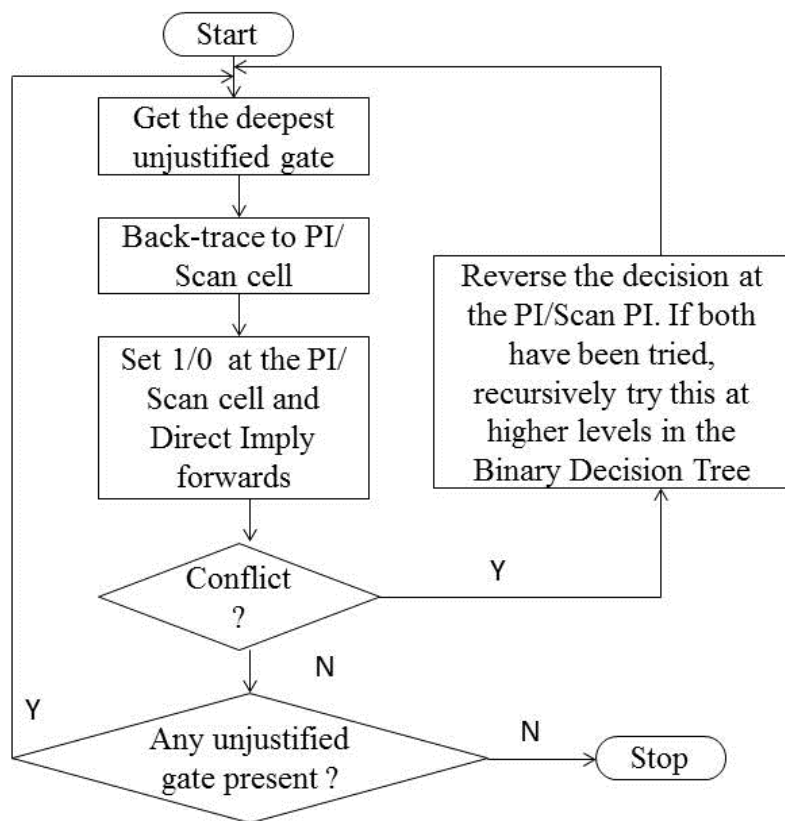
Fig. 9. KLPG Final Justification Algorithm.

1. Create a sorted list (decreasing CO) of unjustified gates. Set level as 0.

2. Get the 1st gate in the original list that is still unjustified. Done when none exist.

3. Back-Trace across time-frames until a PI or a 0th time frame PPI is reached.

4. Set 1/0 at the PI or PPI.

5. Perform Multi-Frame Forward Implications.

6. **If** no conflict is detected

7.   Increase level by 1 and go to step 3.

8. **End If**

9. **Else**

10.   Try the reverse value at the PI or PPI.

11.   Perform Multi-Frame Forward Implications.

12.   **If** no conflict is detected

13.   Increase level by 1 and go to step 3.

14.   **End If**

15.   **Else**

16.   Remove all assignments made at this level and decrease level by 1.

17.   Reverse decisions made higher in the BDD tree and go to step 11.

18.   **End Else**

19. **End Else**

Fig. 10. Final Justification Algorithm.

During Final Justification, when an unjustified gate is encountered, back-trace is performed to a scan cell's 0th time frame, or to a PI in any time frame. In terms of the time frame expanded circuit, Fig. 3, the search must go back to the leftmost set of scan cells, or to a PI in any of the time frames. A value is set on the scan cell/PI and multi-frame direct implication is used to propagate the values forward. Whenever a decision (a logic value on any bit in either pattern) is made at a PI or scan cell, direct implications must be performed to trim the search space. If the gate is now justified, the next unjustified gate is considered, otherwise the search backtracks to reverse the previous decision and the search continued. In case both the options have been tried, the procedure backtracks to the next higher level in the Binary Decision Tree. If both options at the root have been tried and no success was achieved, then the path is dropped. A complete traversal of the entire search tree is infeasible, so a limit is set on the number of unsuccessful attempts at justification. If we exceed this backtrack limit, we drop the path. Such an action can potentially lower the fault coverage. Experiments have been performed with different backtrack limits to show the effect of the backtrack limit on the ATPG process. Hitting the backtrack limit may not result in a loss of coverage, since when a path is dropped, the ATPG tries to generate another long path through the target fault site. This next path can only be as long as the path that was dropped. However, it is still a long path through the target fault site.

In the back-trace, we start from an unjustified gate. This gate is unjustified because one or more of its inputs has the value X in the same time frame. This, in turn, is due to the input gate having one or more X values on its own inputs. This goes on until

we encounter a PI or a PPI. If we reach a PI, we have our decision point. But a PPI is a decision point only if it is in its 0th time frame. If a PPI is not in its 0th time frame, its output is an X because the corresponding PPO value is X in the previous time frame. So, we back-track from the PPO into its own fan-in cone, but this time the process occurs in the previous time frame. This process goes on until we reach either a PI or a PPI in the 0th time frame.

The Final Justification procedure is guided by SCOAP [17] measures. For time frame expansion, we need to calculate the SCOAP measures for each time frame. This is done by calculating the Controllability values for the 0th time frame, then using these values to compute the SCOAP measures for the 1st time frame and so on. Observability values are computed in the last time frame and then propagated backwards until the 0th time frame is reached. This ensures that the search process through the time frames is guided by SCOAP measures that take into account the effect of time frame expansion. This is done in the preprocessing phase. The preprocessing phase also sets the test setup values in the circuit. This is also done in a time frame expanded manner. The values are set in their particular time frames, and are then propagated into the future time frames.

5   COMPACTION FOR PSEUDOFUNCTIONAL TEST

In the domain of scan-based very large scale integrated (VLSI) circuits, the pattern count affects the test cost. This is due to the test application time being proportional to the test set size [21]. Also, if the size of the test set is larger than the memory of the tester, the patterns need to be reloaded, adding an extra cost. A test pattern has 0, 1 or X assigned to each PI and PPI. Compaction takes advantage of unassigned values, i.e. X's to merge together different tests, reducing the total test size. There are two main classes of compaction algorithms for combinational and fully-scanned sequential circuits: static compaction and dynamic compaction.

In Static Compaction, when two patterns are being compacted together, each bit position is compared and the two patterns are replaced by a single pattern. Two patterns can be statically compacted if there are no conflicts at corresponding bit positions of both patterns. A conflict occurs when one pattern has 0 in a particular bit position and the other pattern has a 1. Static Compaction is independent of the test generation and can be performed as a post-processing step. In this research, we have developed a parallel static compaction tool, which we will describe in Section 8.

Dynamic Compaction has already been shown to achieve a higher compaction rate than static compaction, and allows the switching activity to be close to Transition Fault Test patterns [22]. Dynamic Compaction is a part of the test generation process. The classic approach is to generate a pattern for one fault, and then use heuristics to modify the unspecified bits, and drop other detected faults in the fault list via fault simulation. But path delay test has a very low fortuitous detection rate, causing this

approach to be unsuitable. We build upon the previous Dynamic Compaction procedure in CodGen [22]. In this approach, paths were compacted based on their necessary assignments (NAs). This procedure needs to be extended to be able to compact together NAs from multiple time frames. One of the basic problems is the fact that a particular gate can have different values in different time frames as NAs. We need a methodology to efficiently represent such situations and be able to deal with them.

We perform Dynamic Compaction on a path after it has passed the Final Justification procedure. The values discovered during the Final Justification procedure are not NAs, since they are just a set of values that can satisfy the unjustified gates of a single path. A pool of paths, called Path-Pool is maintained, which contains a set of Dynamically Compacted paths. We try to compact the newly justified path with one of the members of Path-Pool. If we do not succeed, we enter this path in the Path-Pool and the ATPG moves on to generating another path through the next target fault site. The Dynamic Compaction approach relies heavily on the Multi-Frame Direct Implication and the Final Justification procedures described earlier.

In a very simple example of Dynamic Compaction, let us consider two paths, Path1 and Path2, through target fault sites A and B respectively. Each path has its own NAs, spread across different time frames. The first step is to put together the NAs of both these paths and check for any direct conflicts in each time frame. A direct conflict is one when there are conflicting values at the output of a particular gate in the same time frame. If there is such a conflict, then these two paths are not compactable with each other. If there is no direct conflict, we perform Multi-Frame Direct Implication to

discover any conflicts. This helps us in detecting a conflict in an early stage in the compaction process. If any conflict is detected, then Path1 and Path2 cannot be compacted with each other. This procedure also helps in finding more NAs which speeds up the next step, Final Justification. If no conflict is detected after the Multi Frame Direct Implication has been performed then we perform the Final Justification procedure described in Section 4.3. This justification is done based on the combined NAs. If this step succeeds, then these two paths are compactable. We discard the assignments discovered during the Final Justification phase and put this path in the Path-Pool. It now contains the NAs of Path1 and Path2, as well as the NAs discovered by implications. Fig. 11 illustrates this.

During the Dynamic Compaction process, there can be cases where the same gate has NAs in more than one time frame. We treat these NAs as independent of each other. The process of performing a compatibility check only between necessary assignments helps in greatly expanding the compaction space without loss of fault coverage. The generation of final test patterns takes place after the test generation and dynamic compaction is completed. This helps us to achieve maximum flexibility for compaction. The Dynamic Compaction algorithm has been integrated with the CodGen code flow.
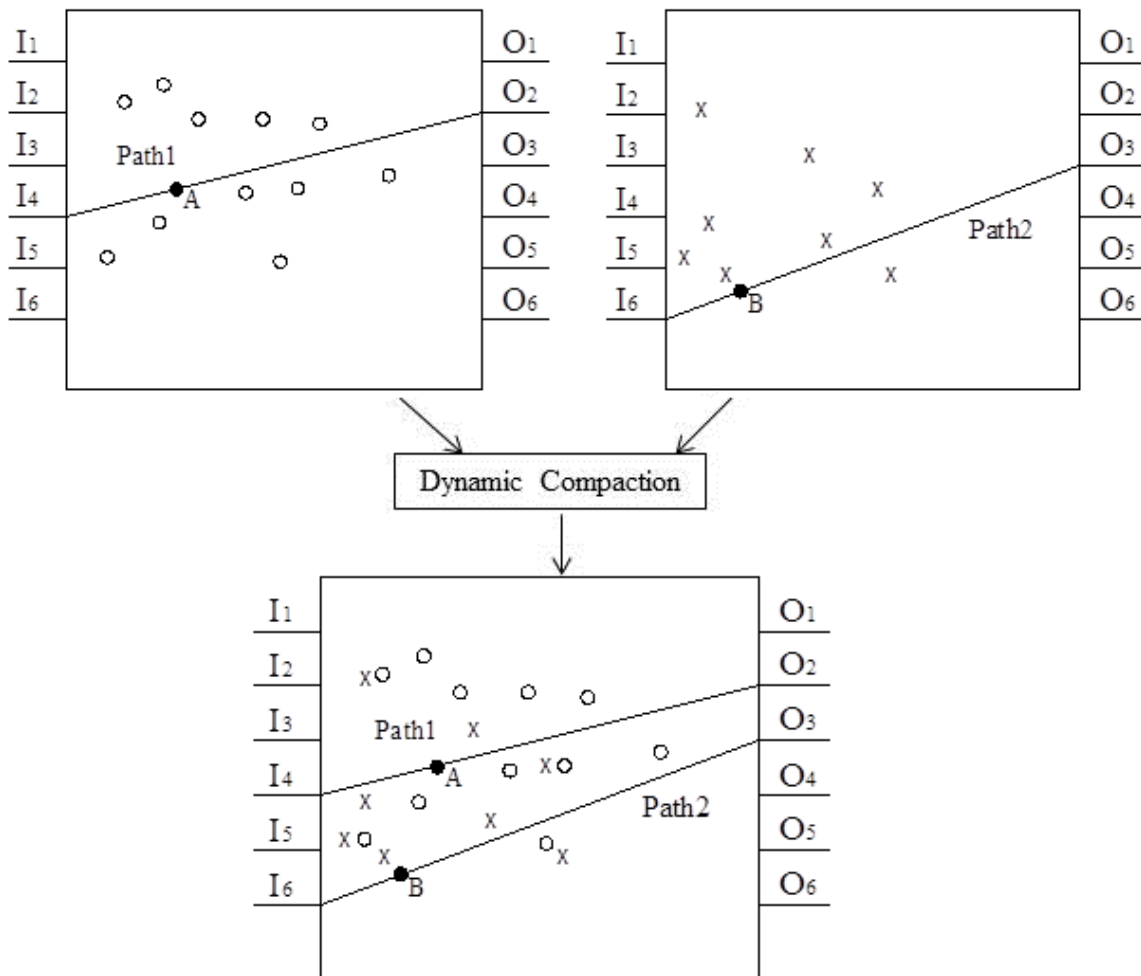
Fig. 11. Dynamic Compaction of Two Paths.

We now describe the Dynamic Compaction procedure in Fig. 12. The high level flow is similar to the original Dynamic Compaction [22]. The internal details, as described above are different, so that the algorithm is able to deal with NAs across multiple time frames as well as generate a vector for the combined paths by traversing multiple time frames in both forward and reverse directions.

1. **If** the Path-Pool is empty, go to step 19.

2. **Else** set the pointer PP at the head of the Path-Pool.

3. **If** there is Direct Conflicts between PP and CP,

4. Move PP to next path in Path-Pool. If no more paths exist, go to step 19.

5. **End If**

6. **Else**

7. Combine NAs from all time frames of PP and CP as TP.

8. Perform Multi-Frame Direct Implication on the combined NAs.

9. **If** conflict occurs

10. Discard TP.

11. Move PP to next path in Path-Pool. If no more paths exist, go to step 19.

12. **End If**

13. **Else**

14. **If** Final Justification (Fig. 10) on TP is successful

15. Discard learning in Final Justification. Replace PP with TP and return.

16. **End If**

17. **End Else**

18. **End Else**

19. Insert CP in Path-Pool and return.

Fig. 12. Dynamic Compaction Algorithm for PFT.

The entire KLPG flow with the Dynamic Compaction procedure is described in Fig. 13.

---

1. Initialize Path-Pool as empty.

2. Use KLPG algorithm (Fig. 7) to generate a complete path, CP including Final Justification (Fig. 10).

3. Discard learning from Final Justification of CP.

4. Call Dynamic Compaction (Fig. 12) for CP.

5. If no more paths can be generated, go to step 6, else go to step 2.

6. Perform Final Justification on each member of Path-Pool to find vectors for them.

Fig. 13. KLPG Algorithm with Dynamic Compaction.

6   RESULTS OF KLPG PATTERN GENERATION

The proposed PFT KLPG pattern generation with integrated Dynamic Compaction has been implemented in Visual C++ and run on an 8 core HP Server with 64-bit Windows 7 Enterprise with a 2.6 GHz AMD Opteron processor and 16 GB of memory. Experiments are performed on the full scan versions of the largest ISCAS89 and ITC99 benchmark circuits.

The experiments were performed for values of K being increased from 1 to 5. This was done in combination with the number of preamble cycles being varied from 0 to 6, in steps of 2. In other words, time frame expansion of 2, 4, 6 and 8 frames was done, and for each frame, KLPG with K = 1, 2, 3, 4 and 5 were observed. In order to study the effect of the Back-Track Limit discussed in Section 4.3, these experiments were performed with two different Back-Track limits.

Low cost testers often have the constraint that PI values must remain constant during the test application process, since the tester has only a few high-speed pins. In this work, we wanted to evaluate the effect of such a constraint. So the experiments were performed keeping PI values constant over all the cycles as well as allowing PI values to change. In the path generation process, the Launch Points were restricted to PPIs, while capture points were restricted to PPOs.

6.1   Path Generation with PIs Held Constant

In order to adhere to the constraints imposed by low cost testers, we assume that the Primary Input (PI) values cannot change during the application of test, since low-cost

testers have only a few high-speed pins. PIs are held constant over all cycles. Similarly, we mask off the Primary Outputs (PO). In other words, we only consider paths that end at a scan cell. Our ATPG has the ability to remove these constraints to evaluate their effect on the path generation process. We generate paths for values of K from 1 to 5. We observe that as the number of time frames increases, the number of sensitizable paths falls. For K = 1 in Fig. 14, there is only a slight decrease for s35932, but a significant drop for s38584 and s38417. For the other circuits, there is a significant drop going from 2 to 4 time frames, and then a gradual drop after that. A similar result is observed for K=5 in Fig. 15, and other values of K. For a constant number of frames, the number of paths rises with K.
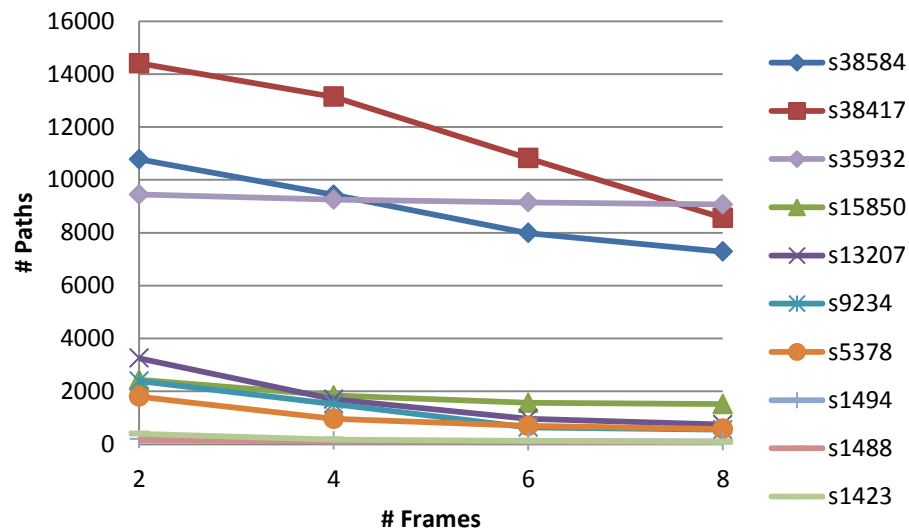
Fig. 14. The Number of Sensitizable Paths for Each Circuit; K=1; Constant PIs.
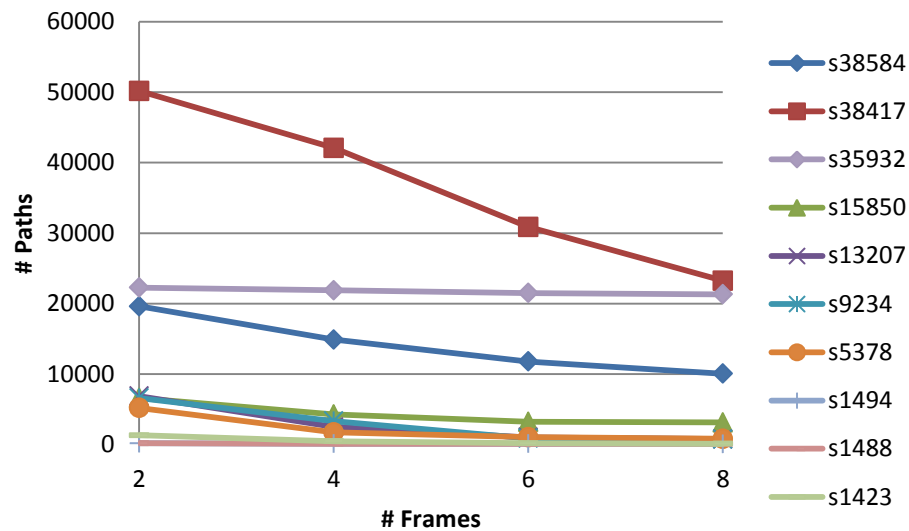
Fig. 15. The Number of Sensitizable Paths for Each Circuit; K=5; Constant PIs.

Relative CPU times are shown in Fig. 16 and Fig. 17. The CPU times are relative to the CPU time for *K*=1 and two time frames, which is the type of KLPG test used in prior work. The CPU time is roughly constant for many circuits. The smaller s5378 has a sublinear CPU time increase. The figures also show that the CPU time rises sublinearly in *K*. The reason that the CPU time rises so slowly with increasing number of time frames is that the decreasing number of sensitizable paths compensates for the increasing effort to find each path.

Fig. 16. CPU Time for K=1, Relative to K=1, Frames=2. PIs Constant.
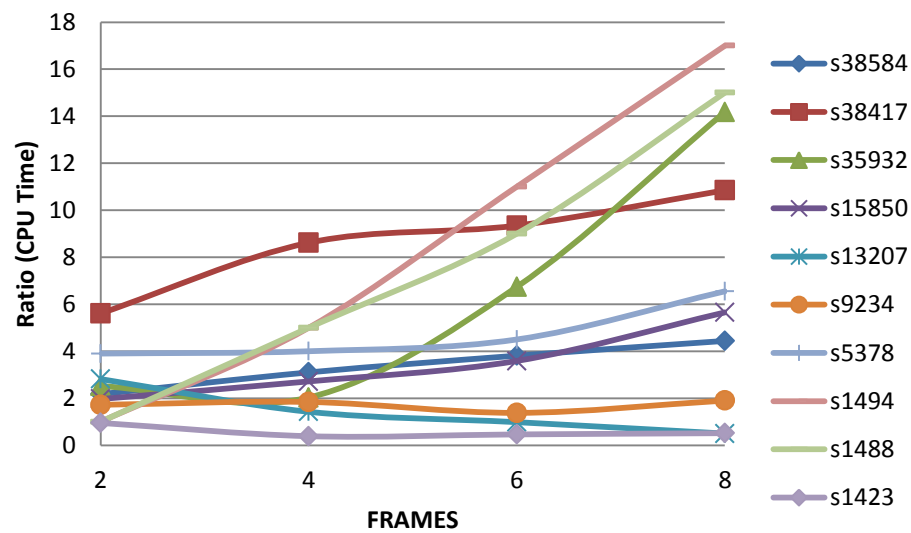
Fig. 17. CPU Time for K=5, Relative to K=1, Frames=2. PIs Constant.

6.2   Path Generation with PIs Not Constant

The impact of holding PIs constant was evaluated by removing this constraint. In practice this would require a tester with high-speed pins on all PIs or corresponding DFT logic. Fig. 18 and Fig. 19 show the results for generating the *K* longest robustly-testable rising and falling paths through each line, under the launch-on-capture constraints, allowing the PIs to have different values in each time frame. Allowing the PIs to have different values can also lead to potentially higher switching activity in the circuit. In these experiments, we do not consider the PIs to be Launch Points and the POs to be capture points. This is done to maintain consistency with Section 6.1. Also, if we require the PIs and the POs to be Launch and Capture points, we would need additional hardware on the testers, which may not be available. We have generated paths, along with Dynamic Compaction of the patterns for 2, 4, 6 and 8 time frame expansions. For each of the time frame, we have taken the value of K = 1, 2, 3, 4 and 5.
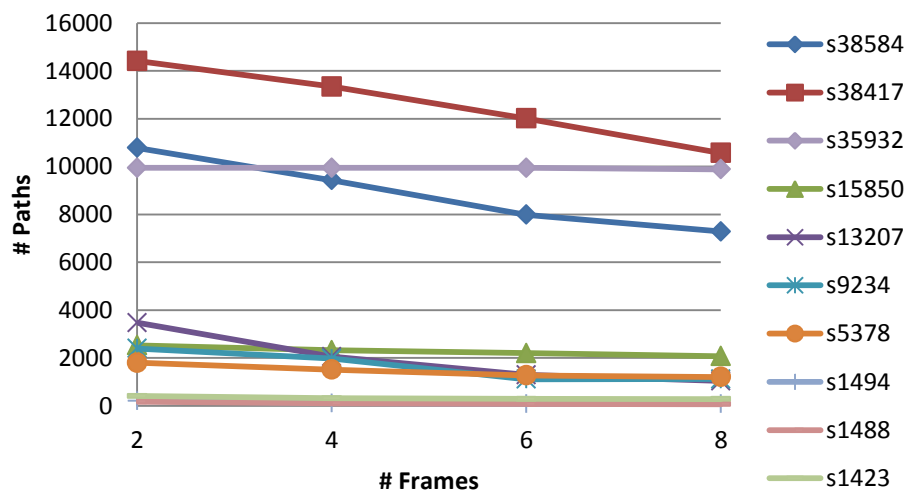
Fig. 18. The Number of Sensitizable Paths for K=1. PIs Not Constant.
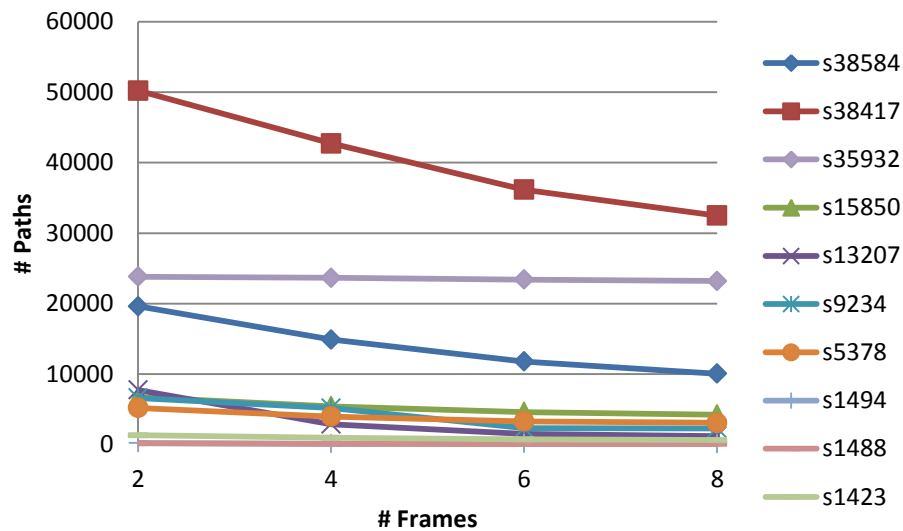
Fig. 19. The Number of Sensitizable Paths for K=5. PIs Not Constant.

In these experiments, we expect an increase in the number of paths, compared to constant PI case, since there are fewer constraints. However, we see a drop in the number of paths in the case of s1494 and s1488. This requires a deeper understanding of the relationship between the Direct Implication and the Final Justification procedures. When we remove the PI fixed value constraint, we are not using Multi-Frame Direct Implication to set the values on the PI. Instead, we are leaving it to the Final Justification procedure to discover these values. Cases arise where the constant value was also the value discovered by Final Justification. However, in some cases, Final Justification hits its Back-Track limit when attempting to justify these paths without the PI constraints. This effect is only visible in circuits where the total number of paths is small and consequently, the dropped paths outnumber the additional paths found. Increasing the

Back-Track limit to 800 found the paths that were dropped without the PI constraint.

These are shown in Fig. 20 and Fig. 21.
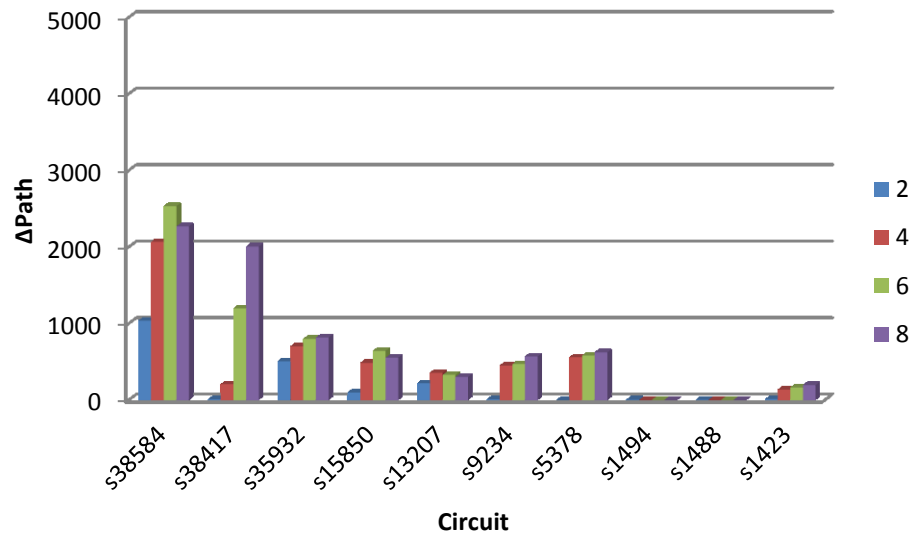
Fig. 20. Increase in # Paths by Removing PI Constraint (BackTrack Limit = 200).
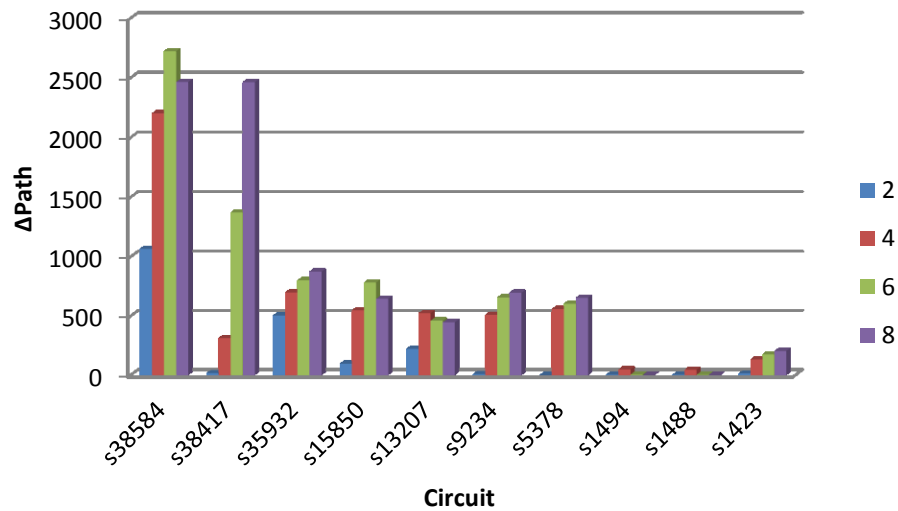
Fig. 21. Increase # Paths by Removing PI Constraint (BackTrack Limit = 800).

The relative CPU times without PI constraints are shown in Fig. 22 and Fig. 23. The CPU times are relative to K=1 and 2 time frames, with no PI constraints. There is more increase in CPU time with time frames than in the case where PIs are held constant, but it is still sublinear. The higher growth rate is because the CPU time for Final Justification rises faster with time frames than does the CPU time for path generation with Direct Implication. In addition, the decrease in paths with increasing number of time frames is only about half as much. This shows that holding the PIs constant for these circuits has a significant impact on the number of paths that can be tested. However, this impact is likely to be much smaller in larger circuits. It must be noted that this time includes the Dynamic Compaction time. This means that the total time is not only dependent on the path generation process, but also includes the time taken to compact each of the generated path with one or more of the previously generated paths. We take the Dynamic Compaction time into account because in a realistic ATPG process, compaction is always a part of the entire pattern generation scheme.

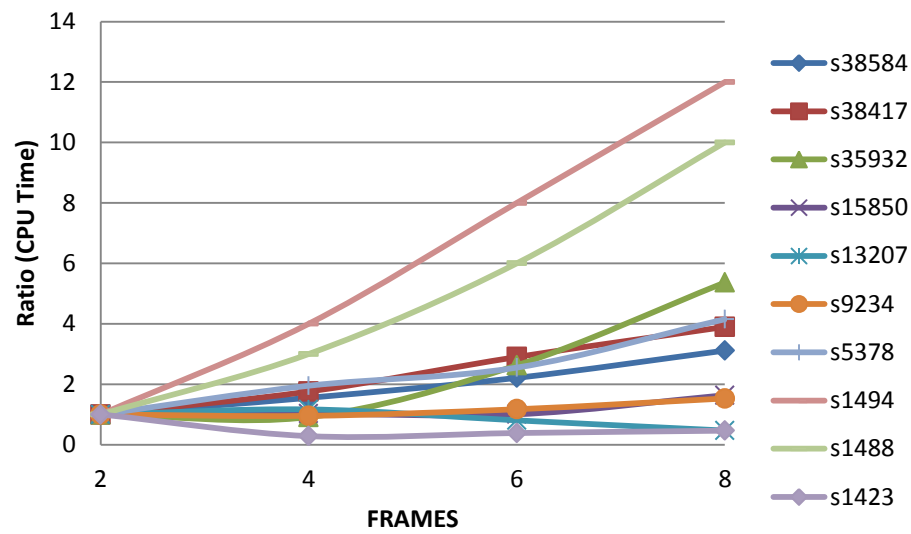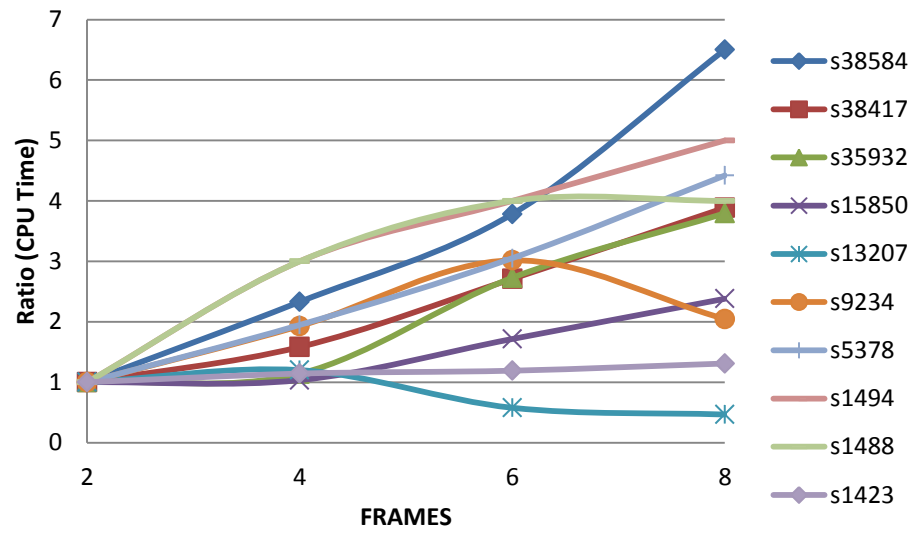**Fig. 22. CPU Time for K=1, Relative to K=1, Frames=2. PIs Not Constant.**
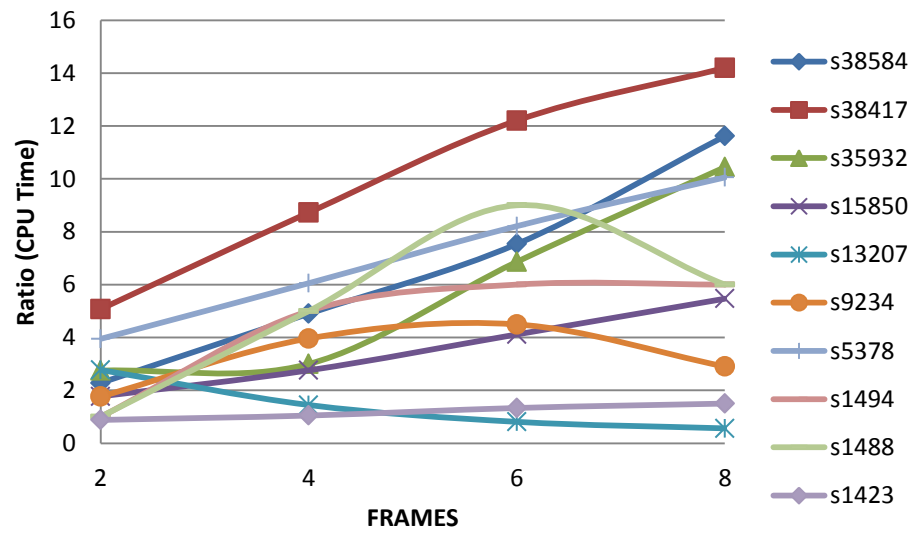
Fig. 23. CPU Time for K=5, Relative to K=1, Frames=2. PIs Not Constant.

6.3   Dynamic Compaction for PFT

In order to reduce the pattern count for PFT, our new dynamic compaction algorithm was used. There are two factors that play a major role in this scenario. One factor is that with the increase in the number of time frames, the care bit density of each pattern increases. This is due to more necessary assignments in the time-frame-expanded circuit, causing the compaction ratio to fall and the pattern count to rise. The second factor that comes into play is that with the increase in the number of time frames, the number of sensitizable paths falls, even when the PIs are allowed to change. This causes the pattern count to go down. The effect of time frame expansion on the pattern count is observed for the PIs held constant and as well as allowing them to change. These are shown in the figures below. As can be seen in the results, the number of patterns is relatively stable as the number of time frames increases, since the reduced compaction ratio roughly balances the decreasing number of paths. This is true both for fixed and changing PIs. The CPU time grows significantly with increasing number of time frames, since the search problem becomes more difficult. The pattern count is shown in Fig. 24 and Fig. 25 for PIs held constant; Fig. 26 and Fig. 27 for PIs changing. The compaction ratios are shown in Fig. 28 and Fig. 29 for PIs held constant; Fig. 30 and Fig. 31 for PIs changing.
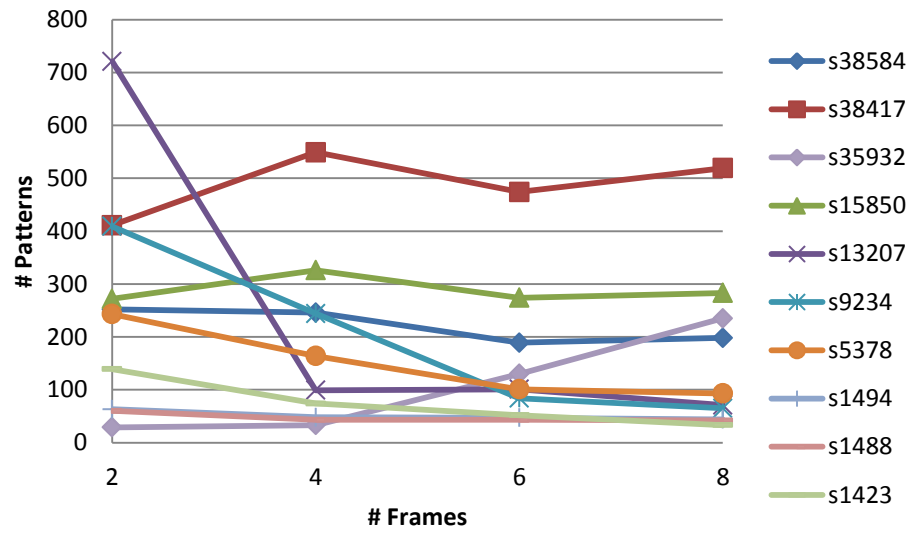
Fig. 24. Pattern Count for K=1, PIs Constant.
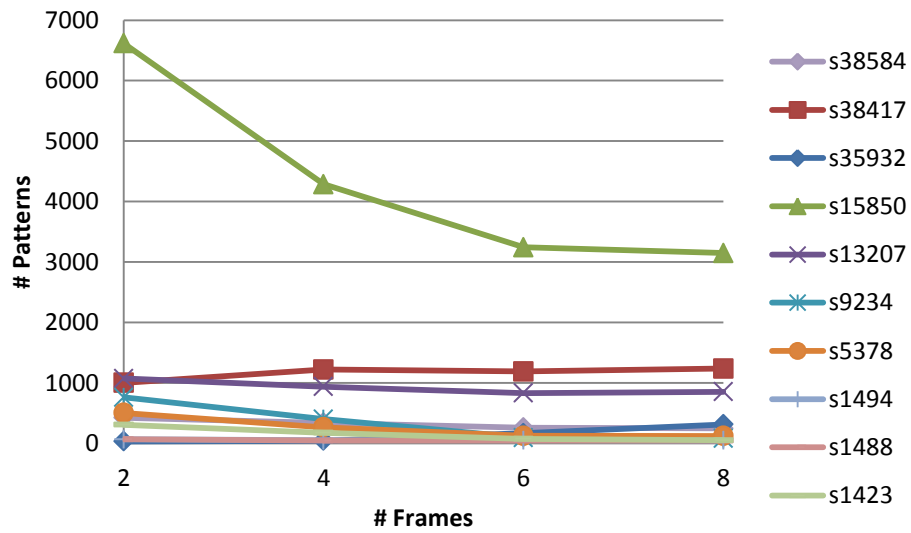
Fig. 25. Pattern Count for K=5, PIs Constant.
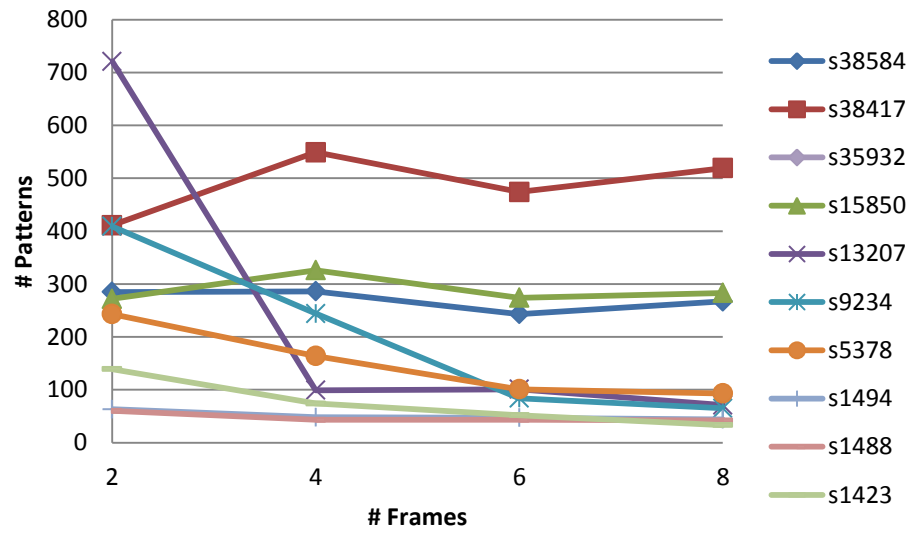
Fig. 26. Pattern Count for K=1, PIs Not Constant.
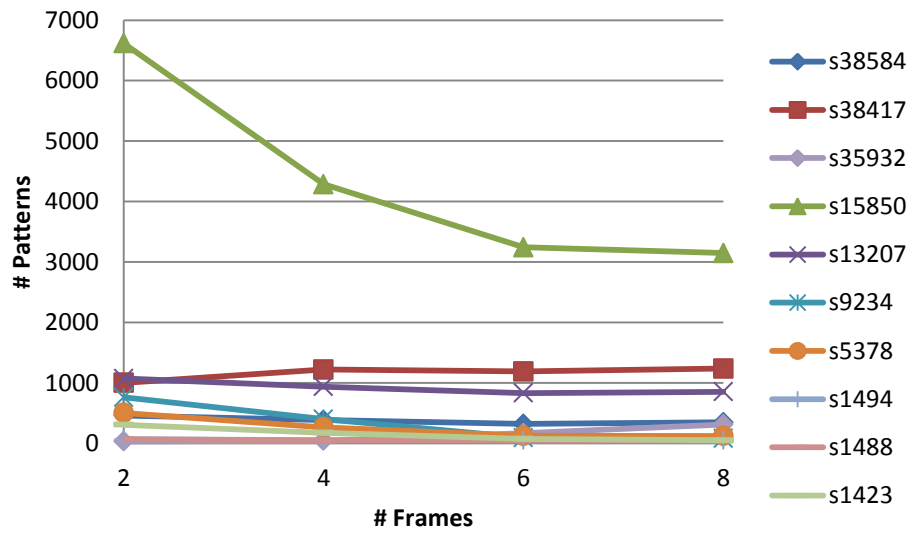
Fig. 27. Pattern Count for K=5, PIs Not Constant.
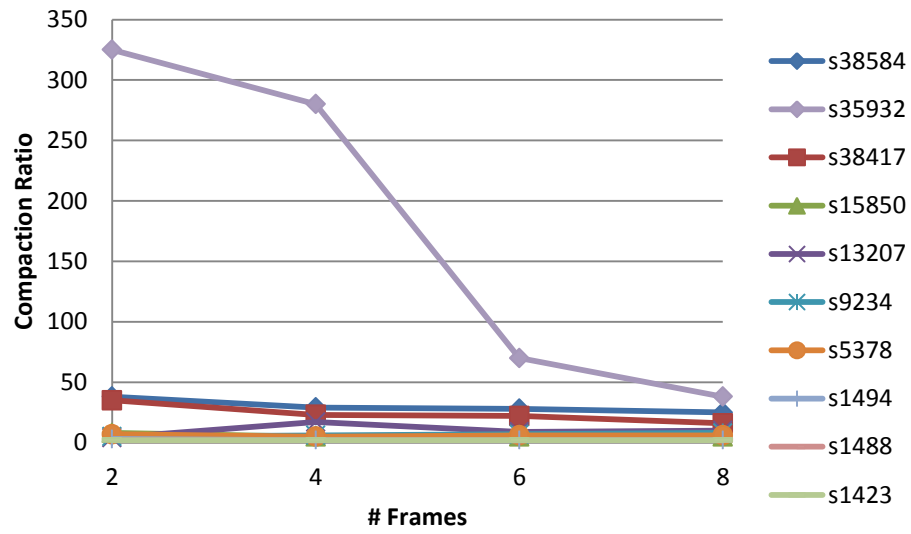
Fig. 28. Compaction Ratio for K=1, PIs Constant.
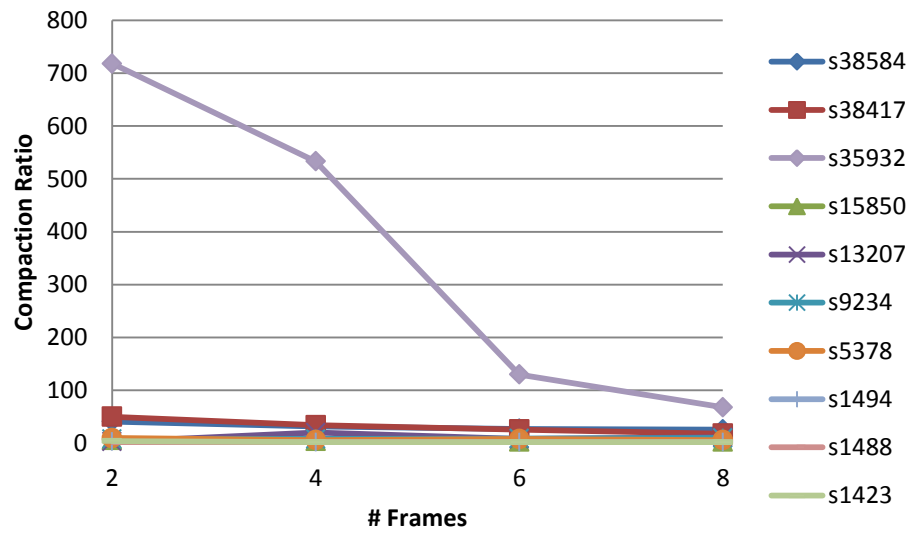
Fig. 29. Compaction Ratio for K=5, PIs Constant.
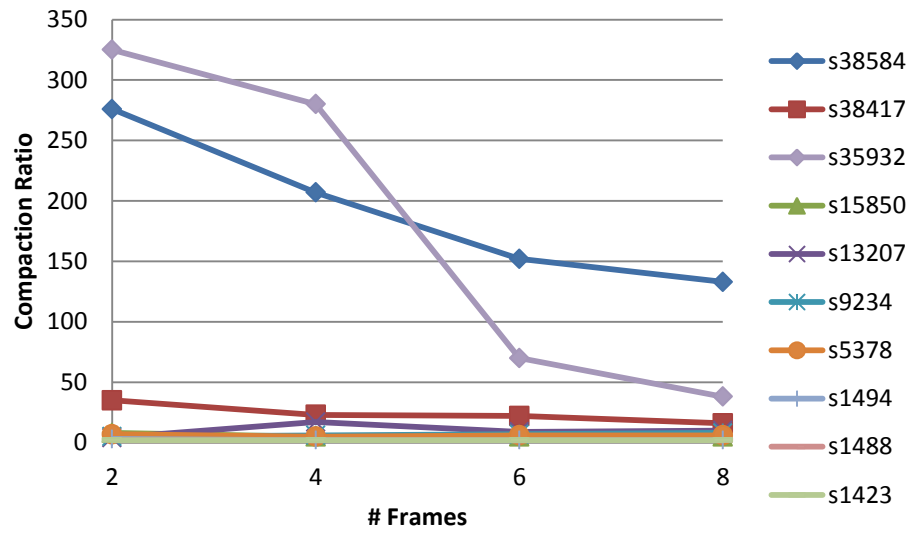
Fig. 30. Compaction Ratio for K=1, PIs Not Constant.
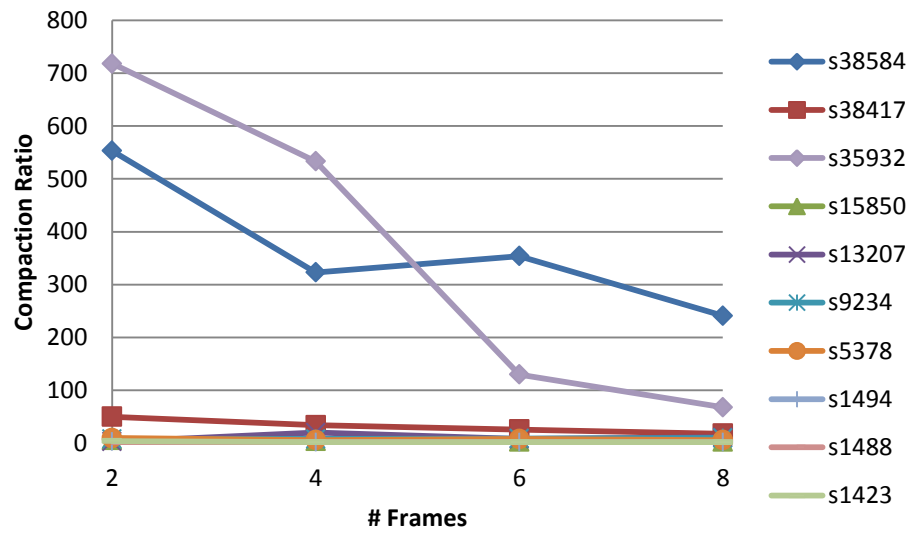
Fig. 31. Compaction Ratio for K=5, PIs Not Constant.

7 POWER GRID MODELING

The scan cycle time is typically an order of magnitude slower than the functional cycle time. The power grid time constant due to off-chip inductance is much longer than the functional clock cycle, so it takes a series of functional clocks before the inductor currents can ramp up to supply the on-chip switching activity. In the meantime, this current must be supplied from on-chip parasitic and decoupling capacitance, causing the supply voltage to droop. This phenomenon has already been explained in Section 2 and 3. One of our main motivations behind PFT was to achieve a high correlation between the power supply behavior in functional and test mode. The introduction of the preamble cycles was meant to facilitate the power grid reaching its functional mode before the at-speed launch and capture cycles were applied.

In this section, we describe a study of the power grid behavior during the PFT process. We also attempt to correlate power grid behavior in the functional mode. Based on a detailed modeling of the power grid and simulations, we would analyze the effect of the PFT patterns on the power supply voltage, as well as the correlation between power grid behavior during the at-speed cycles and the functional state. We also analyze the delay on a path during a traditional two cycle LOC test and compare it with the delay during PFT. In order to make the study more accurate, delay characterization of the standard cells has been done.

7.1   Power Grid Modeling

In order to study the behavior of the power grid, we need to model it in terms of its RLC components [23]. We start with the Design Exchange Format (DEF) files of benchmark circuit s1488. This contains the placement information of the standard cells, as well as the VDD and the GND rails. In addition, there is information about the layout. We parse this information to associate each cell with its VDD and GND rail. Then we model each unit distance on the rail as a RLC block and replace the rails with these blocks. The Verilog file is parsed to set up the input and output connections of each gate, and connect them with their VDD and GND rails. Next the pattern file is scanned. In order to initialize the circuit, we use a MUX to scan in the values in the D Flip Flops (DFF) for the 1st cycle. For subsequent cycles, the select line is flipped so that the DFF are in normal operational mode. Values are initialized on the PIs for all cycles. The VDD and the VSS pins are also modeled as a RLC block. An example is shown in Fig. 32.
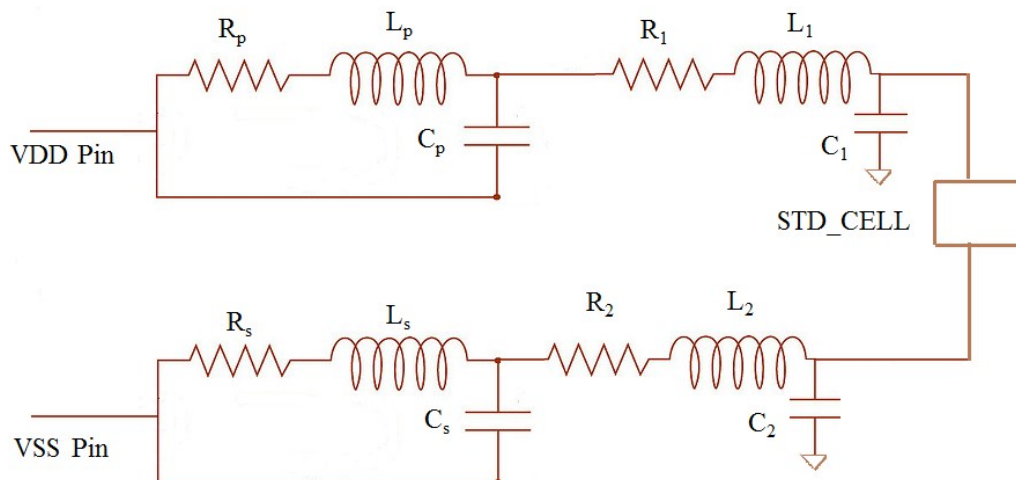
Fig. 32. Power Grid Model.

7.2   Power Grid Simulation

We have used the 45nm PTM High Performance model card from the Predictive Technology Model (PTM) [24] [25] [26] [27] [28]. We have used the NANGATE 45nm standard cell library. We first present a snapshot of voltage variation on the VDD Rail at the point where the RLC unit of the VDD pin feeds the circuit in Fig. 33. In order to study the operating VDD characteristics, waveforms have been observed at other points as well. The ideal VDD is 1.0 V for the technology and model card used. The circuit clock is operated at 2 GHz. The critical path lengths were lesser than 0.5ns, which allowed us to operate at this particular frequency. The power grid, VDD and VSS Pin were modeled with the following parameters:

Resistance = 0.04 Ω/μm.

Inductance = 0.01 nH/μm

Capacitance = 0.2 fF/μm.

R_s = R_p = 0.3 Ω.

L_s = L_p = 0.8 nH.

C_s = C_p = 4 pF

These values were taken from PTM Interconnect Structure 2 parameters for the above mentioned technology. For the simulation, we ran the circuit for 6 half speed preamble cycles (1 GHz) followed by 2 at-speed cycles (2 GHz).
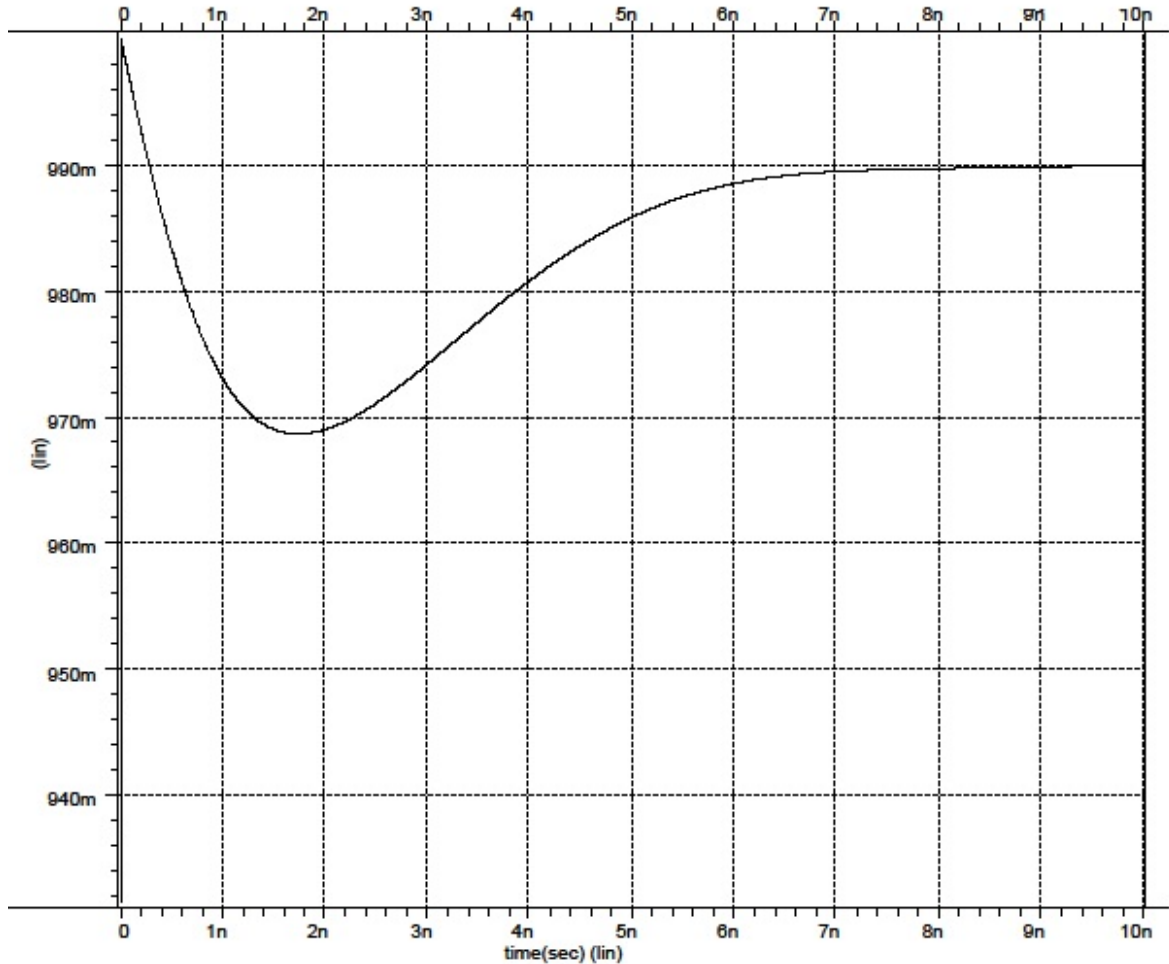
Fig. 33. Voltage Response of Power Grid.

The results show that over the 1st two clock cycles, there is a significant drop in the VDD voltage. This is the time when a normal LOC test would have been applied. During this droop phase, the circuit would have run slower compared to functional mode of operation, since the gates have a lower VDD voltage. This might have caused some critical paths to fail the timing constraints, and thus the chip would be marked as slow. The chip requires a set of preamble cycles before its VDD rails recover to the functional

operating voltage. Our experiments show that the amount of time required for the VDD to recover is fairly independent of the frequency of the preamble cycles. Thus, if we can apply slower clocking during the preamble cycles, we can reduce the number of such cycles required. In this example, we would require 6 at-speed preamble cycles or 3 half-speed preamble cycles. It was also observed that as the circuit size increases, the amount of droop and the recovery time also increase. Thus, it is expected that larger designs will require a higher number (or slower) of preamble cycles than a smaller design. Compared to PFT, a traditional LOC test would have been poorly correlated with functional test, due to different operating conditions. This might have caused a good chip to be marked slow, causing yield loss and loss in profit. Thus PFT plays an important role in improving correlation between functional and test mode while also reducing the test application time.

7.3   Cell Delay Characterization

In order to characterize the individual cell delays, we have simulated their behavior under the robust sensitization criterion. This is because during our ATPG process, we have generated robust paths. The side inputs of the gates have been held at non-controlling values, while the input is allowed to change. In order to make the delay estimate more realistic, the input is fed from an inverter and the logic cell drives an inverter. This more accurately represents circuit conditions, where gates are both driven by and drive other gates. The cells that we have characterized are the ones which are found in the ISCAS89 benchmark circuits. The delay characterization has been performed by varying the VDD in steps of 0.01 V, centered about 1.0 V.
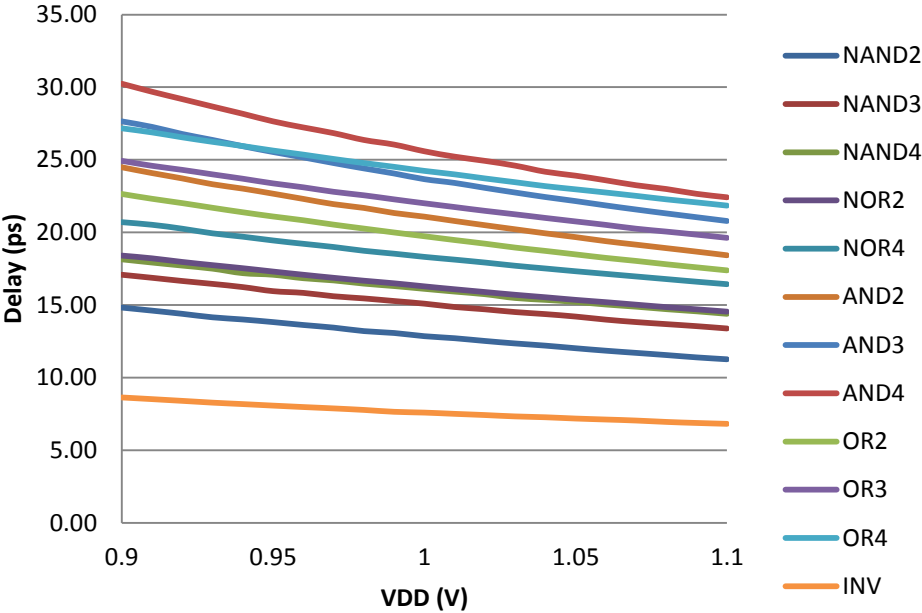
Fig. 34. Rising Delay of Standard Cells.
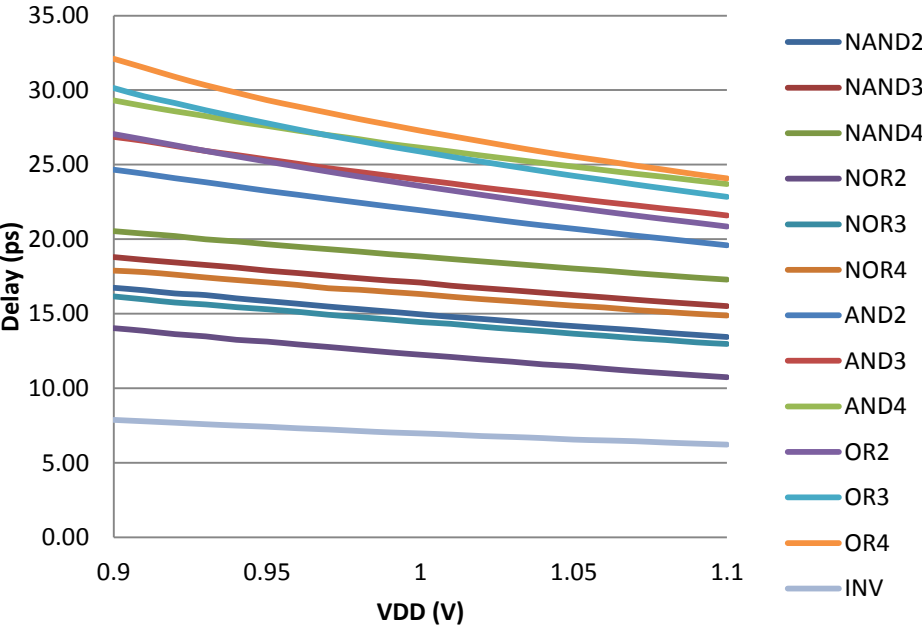
Fig. 35. Falling Delay of Standard Cells.

The delay trends in Fig. 34 and Fig. 35 show that as VDD decreases, the delay of each cell increases. This means that during the droop phase at the start of functional clocking, path delays would be higher than normal.

7.4   Path Delay Comparison

The path delay behavior of the ISCAS89 benchmark circuit s1488 has been studied in detail and is presented here. s1488 offers the advantage of being a small circuit, which can be fully simulated using HSPICE within a reasonable amount of time. Fig. 36 shows the difference in path delay observed for the 40 longest paths (about one-third of the total) during a traditional LOC test and PFT test. This shows that the traditional LOC test finds a path to be slower than PFT. For the purposes of measurement, we have compared the same path under a 2-cycle as well as 8-cycle test. The figure shows the percentage difference in delay predicted by LOC with the delay predicted by an 8-cycle PFT test. It was found that LOC always predicted a higher delay, due to the voltage droop.
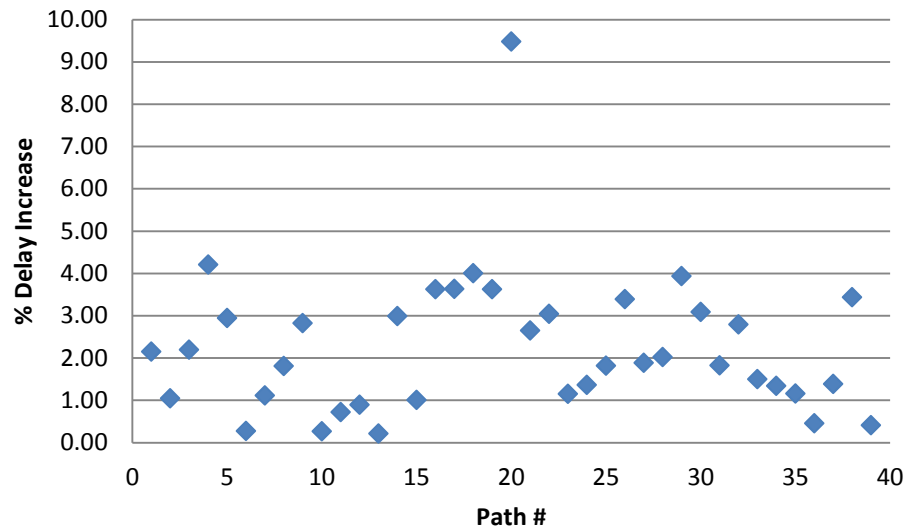
Fig. 36. Delay Difference between LOC and PFT.

The figure shows that there is an average of 3-4 % delay increase when using two-cycle LOC compared to 8-cycle PFT. In the domain of small delay defects, such an error margin can cause a good chip to be labeled as slow. Thus, it is important to be as close to the functional voltage as possible, which is achieved by PFT. The six preamble cycles of the PFT test cause the launch and capture cycles to be applied when the power grid has reached the steady-state, so the circuit delay measured by PFT should closely track that of functional test for this circuit.

It was seen in our experiments that as the circuit size increases, the power grid time constant increases. This would cause industrial designs to have a droop comparable to the ones reported in [6]. Industrial measurements also show a higher inductive response, so more ringing. However, simulating circuits of such magnitude on HSPICE was beyond our cost and time constraints.

8 PARALLEL STATIC COMPACTION

We have considered reducing the compaction time by parallelizing the Static Compaction process. Static Compaction was described in Section 5. A tool has been developed to perform parallel Static Compaction in an efficient manner using the OpenMP model. Static compaction has the advantage that it can be performed as a post-processing step and does not need to be integrated with the ATPG flow. Also, Static Compaction can be performed on patterns that have already been Dynamically Compacted, to further reduce the patter count. Compaction, being a NP complete problem, gives us a lot of scope to use heuristics to achieve near-optimal results [29].

In this section, we evaluate the effect of parallelizing the heuristics on compaction results. The most common heuristics are greedy forward order, reverse order and random access [30]. We also propose a new approach of compaction, using a weighted distribution. We compare the benefits of the different approaches.

8.1 Overview

We take the initial list of patterns. Depending on the approach, we assign a sub-set of the patterns to each of the processors. The processors compact the patterns that they are assigned sequentially and update a Master List containing compacted patterns from each processor. After each of the processors has done their processing, we compact these patterns together. The steps are illustrated in Fig. 37.

:

1. Assign a sub-set of patterns to each processor based on the scheme being implemented.

2. Compact the patterns inside each processor and upload to a Master-List.

3. Compact the master-list.

Fig. 37. Parallel Static Compaction Overview.

The main advantage of parallelization is two-fold. First, each processor is allocated a smaller problem size. Since the complexity of the sequential algorithm is quadratic in terms of the input size, we see a significant reduction in the running time of each processor. Second, by parallelizing the task, we are also able to speed up the entire process and achieve significant reduction in running time.

The major challenge was to break the inherently sequential nature of the compaction algorithms. Previous work [31] uses a methodology where the compaction process uses synchronization at every step. This is inefficient because it limits the amount of speedup that we can achieve. High synchronization overheads reduce the benefits of parallelizing the compaction process. In order to be efficient, we let each processor create its own compacted list of patterns, called Pattern Pool, independent of other processors. This ensures that we avoid per-step synchronization. We have a barrier at the end to cause the master thread to wait for all the processors to finish their execution before running the top-off sequential compaction procedure on the combined list from each processor.

Making the Pattern Pool local to one processor allows us to solve the problem of Concurrent Writes. If we had a globally shared Pattern Pool, it may happen that multiple processors may try to compact a pattern from their assigned patterns into the same pattern in the global Pattern Pool. These patterns may have conflicts among themselves. Making the Pattern Pool local to every processor helps us in avoiding this problem. It also saves the overhead of having an atomic procedure to deal with conflicting concurrent writes.

8.1.1 Forward Order Compaction

We divide the list of patterns into as many segments as there are processors, following the order of patterns. Then we assign each segment to a processor. Each processor compacts the segment of patterns that it has got and updates the master list after which we compact the master list.

8.1.2 Forward Order Compaction with Internal Sorting

This is a modified version of Forward Order Compaction. In this scheme, after we have assigned a set of patterns to each processor, we sort the patterns in each set in descending order based on their care-bit densities. An advantage of this scheme is that instead of sorting the entire list, we are sorting the smaller sets, and sorting in parallel, thus saving time in the sorting operation.

8.1.3 Equal Weighted Bucket Compaction

One of the limitations of the Forward Order compaction schemes is that the time taken by one processor can be much higher than another processor. This can happen

when one processor gets a set of patterns which have very low compatibility. This causes a bottleneck in the parallel computation, since we can finish the parallel processing only as fast as our slowest processor. In order to get around this problem, we would like to distribute the patterns in such a way that each set has nearly equal chances of compaction. This would ensure that the compaction time of the processors have low variance. Weights are assigned based on the care-bit density of the pattern. The idea is based on the fact that the higher the percentage of don't care bits in a pattern, the greater are its chances of compaction. In order to distribute the weights evenly, we have two schemes.

8.1.3.1   Round-robin Weight Distribution

We sort the patterns by weights. Then we divide the patterns into blocks of size equal to number of processors (p). From each block, processor i is assigned pattern I as shown in Fig. 38. The entire chunk is the total set of patterns (n), which is divided into n/p smaller blocks. The numbers inside the smaller squares indicate the processor to which this pattern of the particular block was assigned. n/p = 4 for illustrative purposes.

Fig. 38. Round-Robin Pattern Distribution.

8.1.3.2   Striped Weight Distribution

We sort the patterns by weights and divide the patterns into blocks of size equal number of processors (p). From every odd chunk, processor i is assigned pattern i. From every even chunk, processor i is assigned pattern = number of Processors - i. This is shown in Fig. 39. The entire chunk is the total patterns (n), which is divided into n/p smaller blocks. The numbers inside the smaller squares indicate the processor to which this pattern of a particular block was assigned. n/p = 4 for illustrative purposes.

1	2	...	p	p	......	2	1	1	2	.....	p	p	....	2	1
Block 1				Block 2				Block 3				Block 4			

Fig. 39. Striped Pattern Distribution.

8.2   Parallel Static Compaction Algorithm

In our algorithms, we maintain an individual list of compacted patterns in each processor, called Pattern Pool. This list solely depends on the patterns that were assigned to this processor. So, the compaction process in one processor is completely independent of the compaction process of another processor. This helps us in avoiding communication and synchronization penalties while running the compaction in parallel. After the processors finish, they update the master list of patterns. This is a one-time shared memory access, having lower performance overheads compared to having a globally shared Pattern Pool among the processors. By performing the compaction in parallel, on smaller sets of patterns, we can achieve a high degree of speed-up. The

reason is that the complexity of the sequential algorithm is quadratic in terms of the input size. We run a round of compaction on the master list, because we can further reduce the number of patterns by trying to compact the pattern set from one processor with another. Although this takes more time, we achieve a smaller test set. We can always skip this step if test compaction time is our primary concern. If the number of patterns is our main concern, we perform this final compaction as the amount of extra time is compensated by the time saved in applying the reduced test set on the tester. The algorithm is described in Fig. 40

---

1. Start

2. Divide the Pattern List into segments of size depending on the pattern distribution scheme being used.

3. Assign patterns to processors using one of the pattern distribution schemes.

4. **For** j = 0 to Total Patterns/ No. of Processors *pardo*

5.   **If** the PatternPool is empty

6.   Insert 1st pattern into PatternPool

7.   **Else If** the PatternPool is not empty

8.   **For** k = 0 to current size of PatternPool

9.   **If** jth pattern of Pattern List can be compacted into kth pattern of PatternPool

---

Fig. 40. Parallel Static Compaction Algorithm.

10.   Compact the two patterns and increment Success Counter of kth pattern of Pattern Pool by 1.

11.   **Else** increment Failure Counter of kth pattern of PatternPool by 1.

12.   **End For**

13.   **If** the jth pattern of Pattern List could not be compacted into PatternPool

14.   Add j^{th} pattern of Pattern List to PatternPool and set Success & Failure counter= 0.

15.   **End If**

16.   **End Else**

17. **End For**

18. Add each pattern from PatternPool of each processor to form MasterPatternList.

19. Initialize MasterPatternPool to the 1st pattern in the MasterPatternList, Success Counter & Failure Counter to Success Counter & Failure Counter value of 1st pattern.

20. **For** j = 0 to Total Patterns in MasterPatternList

21.   **For** k = 0 to current size of MasterPatternPool

22.   **If** j^{th} pattern of MasterPatternList can be compacted into k^{th} pattern of MasterPatternPool)

Fig. 40. Continued.

23.   Compact the two patterns and increment Success Counter of kth pattern of Master Pattern Pool by the Success Counter of the jth pattern.

24.   **Else** increment Failure Counter of kth pattern of MasterPatternPool by 1.

25.   **End For**

26.   **If** j^{th} pattern of Pattern List could not be compacted into MasterPatternPool)

27.   Add j^{th} pattern of Master Pattern List to MasterPatternPool and set Success & Failure counter To Success and Failure Counter value of the jth pattern.

28.   **End If**

29. **End For**

30. End

Fig. 40. Continued.

8.3   Experimental Results

We performed all the experiments on a system with 8 AMD Opteron processors with 4 cores each (32 cores total) running at 1.9 GHz with 128 GB of DDR2 RAM. We conducted experiments to study the performance, speedup and strong and weak scaling of the algorithms. For scaling, we use pattern counts of 8,000,000 and 4,000,000 randomly generated patterns. Note that we have chosen a problem size such that the memory used exceeded the cache memory available per core. This is important in smaller problem size because we may observe superlinear scalability. The patterns for

static compaction are generated randomly inside the program using the rand() function. We use the same seed for our experiments to obtain the same set of patterns for each of our algorithms. Each pattern is assigned a care bit density, randomly chosen between 1 and 6%, which is typical in ATPG patterns. Depending on the chosen care bit density, each bit of the pattern is assigned one of the values 0, 1 or X. Each pattern is 100 bits long. All these parameters can be varied in order to model any set of real life patterns. The input sizes start from 1,000,000 and are increased up to 8,000,000 in multiples of 2.

We created a separate class for the timing function, which accurately determines the time by employing the function gettimeofday(). The Timer class object is reset using the function Restart() and the time is obtained by the function GetTime() which reports the time in seconds. Each experiment was repeated 32 times. The execution time was averaged to get more consistent results.

8.3.1   Speedup

Fig. 41 and Fig. 42 show the speedup of the different algorithms.
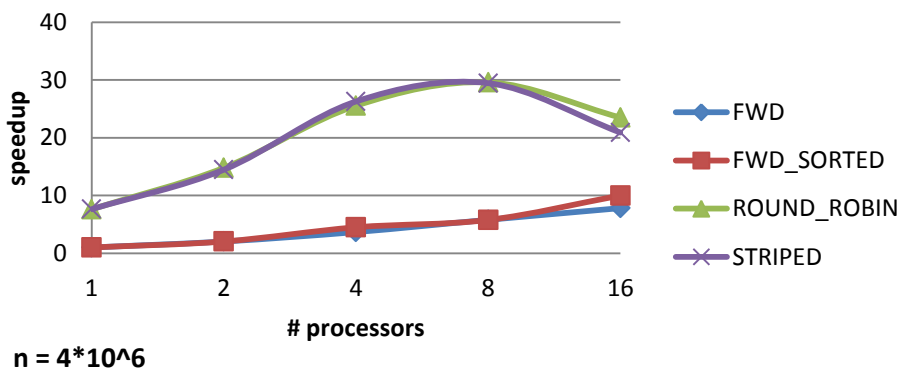
Fig. 41. Speedup of Algorithms vs. # Processors for # Elements = 4*10e6.
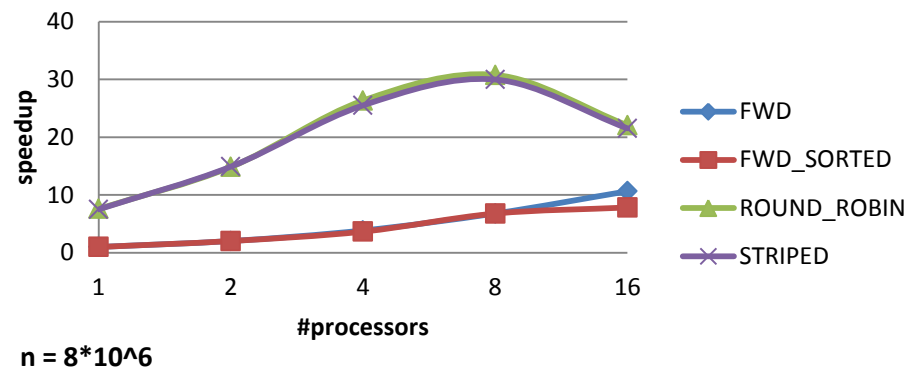
Fig. 42. Speedup of Algorithms vs. # Processors for # Elements = 8*10e6.

We see that Equal Weight Distribution (Round Robin and Striped) has a much higher speedup compared to the Forward Order schemes. Even with the drop in speedup for Equal Weight Distribution schemes in going from 8 to 16 processors, the speedup obtained for these schemes in case of 16 processors is higher than the Forward Order schemes. Also, we see that all these algorithms have a speedup > 1 for all cases. So, we see that parallelizing static compaction has yielded significant benefits in terms of reducing the running time of the process, and that the Equal Weight Distribution approach has a significant speedup for 1 to 16 processors.

8.3.2 Strong Scaling

Fig. 43 and Fig. 44 show the strong scalability of the algorithms. Strong scalability is when the speedup matches the increase in processor count for a fixed problem size. We plot for n = 8,000,000 and n = 4,000,000. We plot only the parallel version for large data sets to avoid effects like caching of the entire data set.
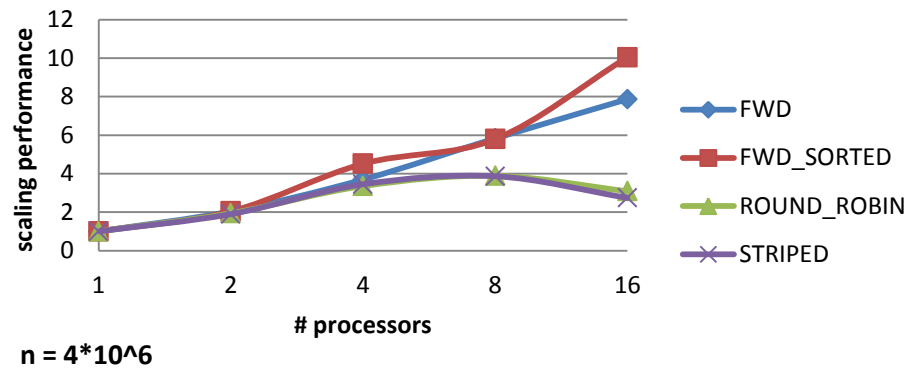
Fig. 43. Strong Scaling for n = 4*10e6.
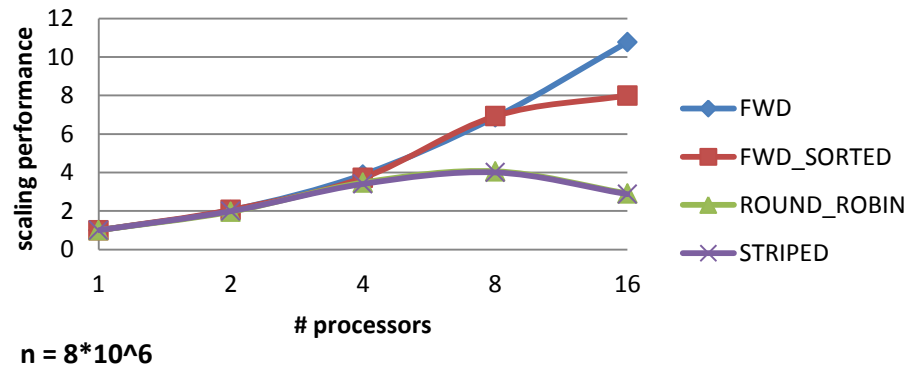
Fig. 44. Strong Scaling for n = 8*10e6.

The large problem size causes the computation cost to dominate over the communication cost, so the speedup is linear or superlinear with processor count (when the data begins to fit into the cache). For smaller overall problem sizes, the communication costs become more important, and the speedup is not as high. The strong scaling of the algorithms is similar to each other because inside the parallel block, the processes are identical. Also, we see that the performance improves as the problem size per processor increases for a higher number of processors.

8.3.3   Weak Scaling

Weak scaling is when the problem size scales with the number of processors. Ideally the total execution time should remain constant. We show the weak scaling plots for n/p = 500000 and n/p = 1000000 for the parallel version. The weak scaling is shown in Fig. 45 and Fig. 46.
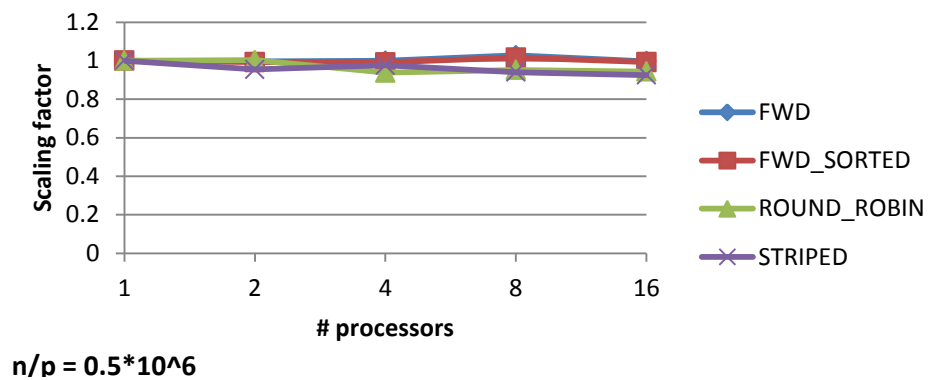
n/p = 0.5*10^6

Fig. 45. Weak Scaling for n/p = 0.5*10e6.
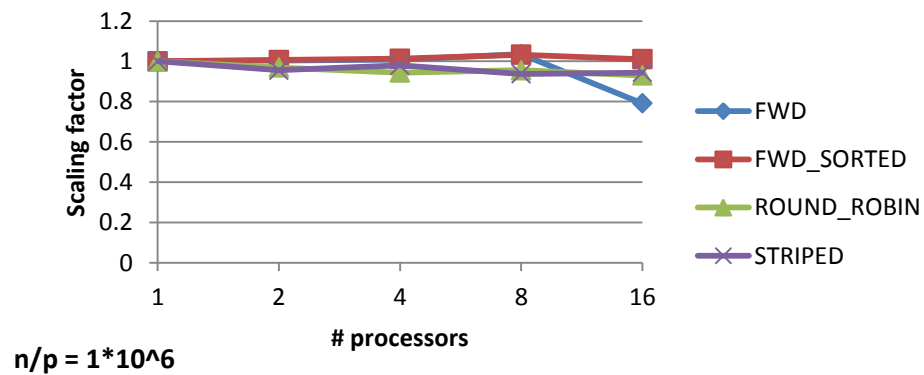
n/p = 1*10^6

Fig. 46. Weak Scaling for n/p = 1*10e6.

.

The plots shows that a fairly ideal weak scaling performance. This shows that for a fixed problem size per processor, the time taken by the parallel algorithms is relatively constant and close to ideal. This can be interpreted as the CPU time per processor remains constant if the problem size per processor remains constant, and the CPU time per processor dominates total time. This is true due to the way static compaction is implemented.

8.3.4 Final Pattern Count

We look at the final pattern count produced by each algorithm. We analyze the effect of increasing the number of processors on final pattern count. Since the purpose of compaction is to reduce the number of patterns that are finally applied to the tester, this is a very important parameter to consider. Fig. 47 and Fig. 48 show a comparative pattern count study.
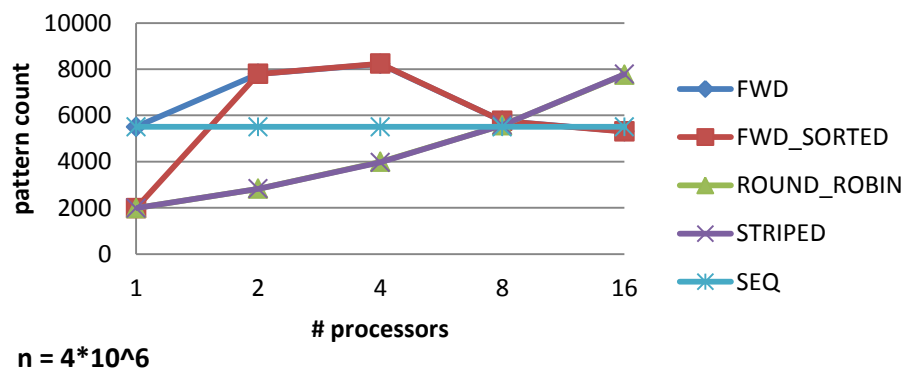
Fig. 47. Pattern Count of Algorithms vs. # Processors for # Patterns = 4*10^6.

n = 8*10^6
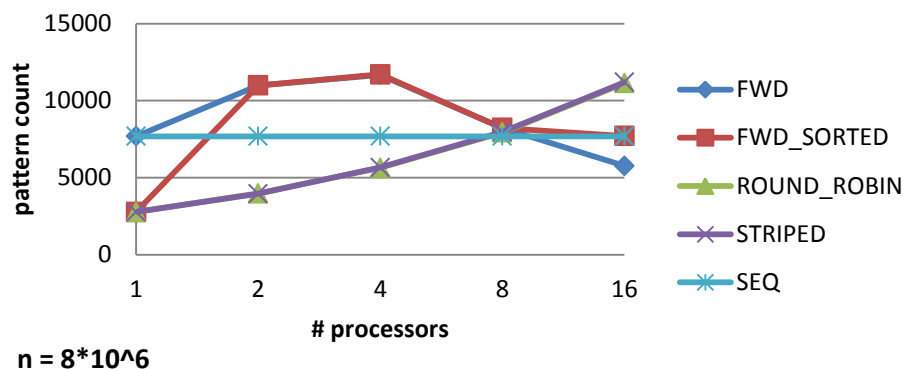
Fig. 48. Pattern Count of Algorithms vs. # Processors for # Patterns = 8*10^6.

These graphs verify our analysis that with increase in the number of processors, we have a smaller set assigned to each processor. Beyond a certain threshold, we suffer reduced compaction among these pattern subsets and a higher total compacted pattern count. This would in turn cause the serial Static Compaction procedure at the end to take more time. The net effect is that we suffer a reduction in speedup beyond a certain point and an increase in pattern count. Despite this, our total running time remains much lower than the sequential running time even with higher number of processors.

Another observation here is that now we have a choice not only in terms of number of processors, but also between algorithms. As we can see, at a higher processor count, the pattern counts of Forward Order schemes are better than the Equal Weight Distribution schemes. Forward Order schemes take less running time than the sequential implementation. However the correct choice would also be dictated by the exact nature of the patterns that we are trying to compact. There is no general answer to the best

solution to the optimization problem that we are facing in this case. Our answer would depend on the situation and the input scenario.

8.4   Discussion

From all the above results, we observe that by parallelizing Static Compaction algorithms we can achieve a significant reduction in running time compared to the best known sequential algorithm. This is possible due to the fact that we have been able to successfully break the patterns into multiple chunks, and utilize each processor to perform compaction independently on that chunk. We have also removed synchronization at every step of the parallel part, so that we can avoid overheads of synchronization and work more efficiently. This is reflected in the speedup that we have obtained with increase in parallelization. We have also seen that the Equal Weight Distribution achieves a better speedup than the Forward Order schemes. This is because the distribution of patterns was done in a manner in which each processor would spend an equal amount of time, rather than waiting on the slowest processor to finish its execution. We have also observed that beyond a certain threshold, a decrease in problem size per processor causes an increase in final pattern count. This occurs as each individual processors have fewer patterns to start. This causes the amount of compaction in the Pattern Pool to decrease. When we run the sequential compaction on the list of patterns that we have obtained by combining the patterns from each processor, we spend a longer time in trying to compact them together. This increases the total running time. In order to avoid this, we need to take care that we give each processor a sufficient number of patterns to compact.

We have already stated that the problem of Static Compaction is NP-Complete. None of the algorithms that we have proposed or the algorithms that are currently being used in the industry, try to achieve the optimal result. Instead, all of them use heuristics to achieve a close to optimal result. This is judged by the pattern count obtained after the compaction process. We have used the pattern count of the sequential algorithm as a benchmark for judging the amount of compaction that the parallel algorithms were able to achieve. We observe that we are able to achieve a significant reduction in pattern count with Equal Weight Distribution. The promising result is that this scheme yields a lower count for a good degree of parallelization. This means that we can use our scheme to reduce the running time without incurring a penalty of increased pattern count.

It is to be noted that we are interested in reducing the pattern count because we are trying to save on the time it takes to apply these patterns on a tester. However, if we use more processors, the amount of time that we need to get these final patterns also reduces. So, the decision must be made whether we want to save time during compaction or during application of these patterns. The choice would be dependent on the priorities that exist at that time.

9   CONCLUSION AND FUTURE WORK

This research addresses the problems of accurate delay test in DSM circuits by targeting resistive open and shorts. The introduction of a Pseudofunctional test (PFT) technique, in which the circuit is initialized to a given state, clocked in normal functional mode for a number of cycles, and then the state is read out, has been shown to have many advantages. One advantage of this approach is that the chip can be clocked in functional mode an order of magnitude faster than the state can be loaded or unloaded. Compared to a traditional scan test, a Pseudofunctional test has the potential to reduce test time and test data volume. The fact that the chip is operating in functional mode increases the correlation between structural and functional test, in terms of functional states, supply noise, power dissipation and signal crosstalk.

In this work, we have proposed the extension of KLPG tests to Pseudofunctional tests with a set of preamble cycles leading to launch-on-capture KLPG tests, in order to control dI/dt noise. Our experiments show that that number of sensitizable paths falls with increasing number of preamble cycles, particularly when the primary inputs are held constant. As a result, the CPU time to generate these tests grows modestly for most circuits. We have also studied the effect of not holding the PIs constant over the test generation. Although letting the PIs take different values in different clock cycles increases the number of sensitized paths, it becomes less important in larger designs. This is primarily due to a very low percentage of pins being PIs and the fact that these circuits have very low care bit density for a single pattern (without compaction).

The memory overhead in implementing PFT in CodGen is mainly in storing the values for additional time frames, including SCOAP values, necessary assignments and temporary values. These grow roughly with the number of time frames, but are still relatively small compared to the memory to hold the circuit description and auxiliary data structures.

We have also developed a dynamic test pattern compaction for PFT. We see that there is a slight reduction in compaction rate for most circuits with increased number of preamble cycles. This is due to the larger number of necessary assignments required to propagate the at-speed test over multiple preamble cycles. However, it is also the case that the decrease in compaction ratio is coupled with a decrease in overall pattern count, so the test generation time does not increase with a higher number of preamble cycles. In addition to the dynamic compaction technique, we have also developed a parallel tool for static compaction. This tool has demonstrated high scalability, with respect to both problem size and number of processors. In environments where dynamic compaction may not be needed, this tool can be used as a post-processing step to the ATPG in order to efficiently reduce the pattern count without loss of test coverage.

One of the questions that arose from this work is that if a path can be sensitized over 6 cycles, but not over 8 cycles, is it a true path. This depends on the circuit operation, and whether the state found searching back 6 cycles is a reachable state. From a test generation perspective, this is similar to the situation of faults that can be tested under launch-on-shift constraints, but not launch-on-capture.

Profiling of the experiments indicates that the vast majority of the time is spent in Final Justification. This indicates that a more efficient search algorithm than PODEM must be used. This is important because the same Final Justification algorithm is being used in justification of a complete path, as well as during the dynamic compaction process. A faster algorithm would benefit the entire ATPG process. In addition to a faster justification procedure, the low-cost fault coverage metric, where paths with large slack are targeted by transition fault test, needs to be integrated with PFT. The metric must be able to take into account the effect of the preamble cycles in addition to process variation. An area that needs investigation is applying multiple at-speed cycles, that is, the longest path is tested across several cycles, which can target latch time borrowing and flip-flop metastability.

REFERENCES

[1] Semiconductor Industries Association, *International Technology Roadmap for Semiconductors*. San Jose, CA: Semiconductor Industries Association, 2009.

[2] H. Liu and M. S. Hsiao, "Constrained ATPG for broadside transition testing," in *Proc. IEEE Intl. Symp. DFT VLSI Syst.*, Nov. 2003, pp. 175-182.

[3] P. Gupta and M. S. Hsiao, "High quality ATPG for delay defects," in *Proc. IEEE Intl. Test Conf.*, Sep. 2003, pp. 584-591.

[4] N. Touba, "Survey of test vector compression techniques," *IEEE Trans. Design Test Comput.*, vol. 23, no. 4, pp. 294-303, Jul. 2006.

[5] B. Nadeau-Dostie, K. Takeshita, and J. Cote, "Power-aware at-speed scan test methodology for circuits with synchronous clocks," in *Proc. IEEE Intl. Test Conf.*, Sep. 2008, pp. 1-10.

[6] P. Pant and J. Zelman, "Understanding power supply droop during at-speed scan testing," in *Proc. IEEE VLSI Test Symp.*, May 2003, pp. 227-232.

[7] Z. Zhang, S. M. Reddy, and I. Pomeranz, "On generating pseudo-functional delay fault tests for scan designs," in *Proc. IEEE VLSI Test Symp.*, May 2005, pp. 398-405.

[8] Y. Lin, F. Lu, and K. Cheng, "Pseudofunctional testing," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 25, no. 8, pp. 1535-1546, Aug. 2006.

[9] F. Yuan and Q. Xu, "On systematic illegal state identification for pseudo-functional testing," in *Proc. Des. Autom. Conf.*, Jul. 2009, pp. 702-777.

[10] W. Wu and M. S. Hsiao, "Mining sequential constraints for pseudo-functional testing," in *Proc. Asian Test Symp.*, Oct. 2007, pp. 19-24.

[11] X. Liu and M. S. Hsiao, "A novel transition fault ATPG that reduces yield loss," *IEEE Trans. Design Test Comput.*, vol. 22, no. 6, pp. 576-584, Nov. 2005.

[12] M. Syal, K. Chandrasekar, V. Vimjam, and M. S. Hsiao., "A study of implication based pseudo functional testing," in *Proc. IEEE Intl. Test Conf.*, Sep. 2006, pp. 1-10.

[13] H. Liu, H. Li, Y. Hu, and X. Li, "A scan-based delay test method for reduction of overtesting," in *Proc. IEEE Intl. Symp. Electronic Design, Test Applications*, Jan. 2008, pp. 521-526.

[14] L. T. Wang, C. E. Stroud, and N. A. Touba, *System-on-Chip Test Architectures: Nanometer Design for Testability*. Burlington, VT: Elsevier, 2008.

[15] W. Qui and D. M. H. Walker, "An efficient algorithm for finding the k longest testable paths through each gate in a combinational circuit," in *Proc. IEEE Intl. Test Conf.*, Sep. 2003, pp. 223-231.

[16] W. Qiu, X. Lu, D. M. H. Walker, and W. Shi, "A statistical fault coverage metric for realistic path delay faults," in *Proc. IEEE VLSI Test Symp.*, May 2004, pp. 27-42.

[17] L. H. Goldstein and E. L. Thigpen, "SCOAP: Sandia controllability / observability analysis program," in *Proc. IEEE-ACM Des. Autom. Conf.*, Jun. 1980, pp. 190-196.

[18] H. Fujiwara, "FAN: A fanout-oriented test pattern generation algorithm," in *Proc.*

ISCAS 85, Jun. 1985, pp. 671-674.

[19] H. Fujiwara and T. Shimono, "On the acceleration of test generation algorithms," *IEEE Trans. Computers*, vol. C-32, pp. 1137-1144, Dec. 1983.

[20] P. Goel, "An implicit enumeration algorithm to generate tests for combinational logic circuits," *IEEE Trans. Computers*, vol. 30, no. 3, pp. 215-222, Mar. 1981.

[21] I. Hamzaoglu and J. H. Patel, "Test set compaction algorithms for combinational circuits," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 19, no. 8, pp. 957 - 963, Aug. 2000.

[22] Z. Wang and D. M. H. Walker, "Dynamic compaction for high quality delay test," in *Proc. VLSI Test Symp.*, May 2008, pp. 243-248.

[23] M. Nourani, M. Tehranipoor, and N. Ahmed, "Pattern generation and estimation for power supply noise analysis," in *Proc. IEEE VLSI Test Symp.*, May 2005, pp. 439-444.

[24] S. Zhao, K. Roy, and C.K. Koh, "Estimation of inductive and resistive switching noise on power supply network," in *Proc. Intl. Conf. Comput. Des.*, Sep. 2000, pp. 65-72.

[25] W. Zhao and Y. Cao, "New generation of predictive technology model for sub-45nm early design exploration," *IEEE Trans. Electron Devices*, vol. 53, no. 11, pp. 2816-2823, Nov. 2006.

[26] Y. Cao, T. Sato, D. Sylvester, M. Orshansky, and C. Hu, "New paradigm of predictive MOSFET and interconnect modeling for early circuit design," in *Proc.*

Custom Integr. Circuits Conf., Jun. 2000, pp. 201-204.

[27] A. Balijepalli, S. Sinha, and Y. Cao, "Compact modeling of carbon nanotube transistor for early stage process-design exploration," in *Proc. Intl. Symp. Low Power Electronics Design*, Aug. 2007, pp. 2-7.

[28] Nanoscale Integration and Modeling (NIMO) Group. Predictive Technology Model. [Online]. http://ptm.asu.edu/

[29] J. Wang Z. Yue, X. Lu, W. Qiu, W. Shi, and D. M. H. Walker, "A vector-based approach for power supply noise analysis in test compaction," in *Proc. IEEE Intl. Test Conf.*, Sep. 2005, pp. 517-526.

[30] J. Wang, X. Lu, W. Qiu, Z. Yue, S. Fancler, W. Shi, and D. M. H. Walker, "Static compaction of delay tests considering power supply noise," in *Proc. IEEE VLSI Test Symp.*, May 2005, pp. 235-240.

[31] X. Cai, P. Wohl, J. A. Waicukausk, and P. Notiyath, "Highly efficient parallel ATPG based on shared memory," in *Proc. IEEE Intl. Test Conf.*, Nov. 2010, pp. 1-7.

APPENDIX

TABLE I
KLPG RESULTS, K=1

Circuit	Params	PI Constant				PI Not Constant			
		FRAMES				FRAMES			
		2	4	6	8	2	4	6	8
s38584	#Paths	9742	7367	5451	5016	10777	9432	7983	7284
	#Patterns	252	246	189	198	285	286	243	267
	Time	973	1508	2150	3029	1214	2828	4592	7894
s38417	#Paths	14405	13139	10818	8562	14413	13343	12013	10566
	#Patterns	411	549	474	519	409	554	508	525
	Time	3213	5648	9315	12519	3393	5369	9196	13188
s35392	#Paths	9442	9248	9145	9070	9946	9946	9941	9884
	#Patterns	29	33	130	235	36	48	65	74
	Time	940	858	2467	5045	1068	1218	2914	4053
s15850	#Paths	2434	1840	1562	1519	2530	2323	2201	2070
	#Patterns	272	326	274	283	272	222	229	221
	Time	250	256	251	409	260	269	446	619
s13207	#Paths	3253	1699	965	741	3469	2053	1291	1039
	#Patterns	721	99	101	71	813	106	99	65
	Time	163	190	131	76	215	258	124	100
s9234	#Paths	2385	1524	641	550	2393	1972	1109	1117
	#Patterns	409	244	84	65	402	349	196	227
	Time	233	222	273	356	239	461	721	489
s5378	#Paths	1801	958	694	579	1801	1514	1273	1200
	#Patterns	243	164	101	93	238	175	163	152
	Time	20	39	51	83	19	37	58	84
s1494	#Paths	193	129	113	107	197	112	77	77
	#Patterns	63	48	47	44	65	50	33	34
	Time	1	4	8	12	1	3	4	5
s1488	#Paths	167	109	104	94	170	102	70	68
	#Patterns	60	43	43	42	63	48	31	32
	Time	1	3	6	10	1	3	4	4
s1423	#Paths	397	180	128	86	407	312	290	279
	#Patterns	139	74	52	33	140	123	93	83
	Time	39	11	15	18	42	48	50	55

TABLE II
KLPG RESULTS, K=2

Circuit	Params	PI Constant FRAMES				PI Not Constant FRAMES			
		2	4	6	8	2	4	6	8
s38584	#Paths	13258	9442	6539	6055	14755	12412	10167	9038
	#Patterns	325	314	230	233	360	356	295	318
	Time	1173	2156	3056	3720	1944	4056	7630	12382
s38417	#Paths	25400	22793	18019	13970	25414	23093	20236	17883
	#Patterns	580	758	700	776	580	751	702	802
	Time	6169	11925	16028	20430	6964	10792	16939	23813
s35392	#Paths	20255	19967	19680	19488	21534	21534	21534	21389
	#Patterns	33	39	156	296	48	59	67	96
	Time	1733	1712	4903	11162	1925	2689	4949	8535
s15850	#Paths	3810	2735	2167	2089	3913	3452	3155	2943
	#Patterns	461	506	437	451	468	380	382	386
	Time	285	335	378	590	299	401	599	865
s13207	#Paths	4490	1984	1024	780	4872	2459	1488	1201
	#Patterns	1144	114	112	70	1254	123	113	69
	Time	234	207	157	77	298	207	145	113
s9234	#Paths	3753	2290	800	680	3763	3104	1587	1569
	#Patterns	544	316	90	75	550	459	244	268
	Time	285	301	292	384	281	604	860	608
s5378	#Paths	2996	1323	915	740	2996	2422	2048	1916
	#Patterns	327	208	115	109	320	246	238	223
	Time	35	60	67	110	32	66	99	122
s1494	#Paths	202	133	115	108	206	116	115	79
	#Patterns	64	49	48	44	66	51	48	35
	Time	1	4	9	14	1	3	6	5
s1488	#Paths	185	118	110	100	188	112	76	76
	#Patterns	62	45	45	43	65	49	34	36
	Time	1	4	8	12	1	4	4	5
s1423	#Paths	690	296	184	118	700	503	463	418
	#Patterns	200	110	68	43	203	151	131	109
	Time	41	12	16	18	40	43	49	56

TABLE III
KLPG RESULTS, K=3

Circuit	Params	PI Constant FRAMES				PI Not Constant FRAMES			
		2	4	6	8	2	4	6	8
s38584	#Paths	15284	10328	6887	6294	17116	13737	11017	9617
	#Patterns	349	333	257	239	394	367	317	334
	Time	1783	2666	3497	4150	2387	4525	8629	12436
s38417	#Paths	34686	30332	23212	17664	34707	30757	26474	23536
	#Patterns	733	951	891	966	724	942	884	1013
	Time	10195	16709	22353	28002	9776	16519	25653	33490
s35392	#Paths	21458	21074	20690	20498	22738	22720	22589	22380
	#Patterns	31	41	164	310	74	41	70	97
	Time	1997	1750	6029	12805	2394	2701	6029	10226
s15850	#Paths	4853	3294	2572	2477	4964	4239	3772	3500
	#Patterns	690	658	583	597	693	526	494	507
	Time	343	455	556	849	345	534	862	1022
s13207	#Paths	5352	2192	1032	786	5908	2658	1508	1214
	#Patterns	1495	114	119	69	1638	123	119	68
	Time	306	214	146	84	384	284	156	120
s9234	#Paths	4792	2748	867	736	4802	3915	1891	1860
	#Patterns	663	352	94	74	658	530	270	281
	Time	323	404	325	410	319	733	955	616
s5378	#Paths	3937	1538	1018	806	3937	3151	2656	2495
	#Patterns	415	241	126	116	408	313	298	286
	Time	54	78	84	127	50	91	127	163
s1494	#Paths	202	133	115	108	206	116	115	79
	#Patterns	66	50	48	44	67	51	48	35
	Time	2	5	10	16	2	4	10	6
s1488	#Paths	201	120	112	103	191	113	77	77
	#Patterns	69	45	45	43	65	50	34	35
	Time	1	5	8	13	1	4	8	13
s1423	#Paths	929	378	193	130	941	679	599	511
	#Patterns	235	139	71	48	237	202	146	122
	Time	39	13	17	20	40	13	51	60

TABLE IV
KLPG RESULTS, K=4

Circuit	Params	PI Constant FRAMES				PI Not Constant FRAMES			
		2	4	6	8	2	4	6	8
s38584	#Paths	16505	10815	7033	6396	18615	14540	11543	9936
	#Patterns	390	333	255	249	428	382	329	335
	Time	1927	2483	3364	4265	2569	5646	8864	15861
s38417	#Paths	42714	36539	27317	20724	42745	37085	31563	28343
	#Patterns	887	1115	1048	1110	880	1101	1019	1166
	Time	13989	22511	27977	32992	14189	25464	33229	42521
s35392	#Paths	21856	21472	21088	20896	23392	23232	22999	22784
	#Patterns	31	41	165	311	43	73	66	96
	Time	2356	2007	6104	13014	2586	3385	7059	10831
s15850	#Paths	5798	3806	2929	2836	5798	4916	4270	3950
	#Patterns	892	806	712	741	897	656	601	631
	Time	415	573	679	1135	396	625	977	1222
s13207	#Paths	6130	2365	1038	786	6130	2789	1521	1220
	#Patterns	1779	123	124	69	1779	137	124	66
	Time	381	225	154	82	381	293	176	114
s9234	#Paths	5784	3101	894	770	5792	4616	2155	2100
	#Patterns	697	384	93	75	704	580	284	294
	Time	369	397	325	423	361	838	977	648
s5378	#Paths	4738	1659	1049	824	4738	3753	3139	2962
	#Patterns	471	262	126	120	467	382	354	344
	Time	69	81	87	125	68	115	144	180
s1494	#Paths	202	133	115	108	206	116	115	79
	#Patterns	65	50	48	46	66	51	48	35
	Time	2	5	10	16	2	4	5	6
s1488	#Paths	189	121	113	104	192	114	77	77
	#Patterns	64	45	46	44	66	51	34	35
	Time	1	5	8	13	1	4	8	13
s1423	#Paths	1139	439	197	135	1153	828	695	580
	#Patterns	272	162	73	50	278	234	165	137
	Time	37	14	17	20	40	41	55	65

TABLE V
KLPG RESULTS, K=5

Circuit	Params	PI Constant FRAMES				PI Not Constant FRAMES			
		2	4	6	8	2	4	6	8
s38584	#Paths	17314	11031	7072	6428	19621	14909	11799	10076
	#Patterns	422	338	258	245	459	382	318	346
	Time	2117	3015	3708	4319	2769	5947	9147	14099
s38417	#Paths	50172	42085	30885	23255	50209	42721	36168	32496
	#Patterns	999	1220	1187	1234	992	1206	1144	1307
	Time	18024	27682	30033	34861	17212	29595	41407	48168
s35392	#Paths	22274	21890	21506	21314	23810	23650	23417	23202
	#Patterns	31	41	165	311	43	73	66	96
	Time	2367	1908	6333	13326	2933	3203	7326	11152
s15850	#Paths	6618	4284	3241	3147	6737	5440	4610	4263
	#Patterns	1071	935	831	852	1077	785	718	735
	Time	492	679	897	1413	461	717	1070	1421
s13207	#Paths	6886	2513	1042	786	7760	2893	1526	1225
	#Patterns	2068	123	127	69	2254	141	125	66
	Time	459	232	160	81	596	311	174	121
s9234	#Paths	6605	3342	905	779	6613	5187	2330	2297
	#Patterns	759	399	92	77	759	606	289	311
	Time	401	429	321	443	422	947	1074	693
s5378	#Paths	5214	1731	1052	826	5214	3988	3300	3109
	#Patterns	501	263	122	121	398	263	369	364
	Time	78	80	90	131	75	115	156	191
s1494	#Paths	202	133	115	108	206	116	115	79
	#Patterns	65	50	48	46	66	51	48	35
	Time	1	5	11	17	1	5	6	6
s1488	#Paths	189	121	113	104	189	121	77	77
	#Patterns	65	45	46	44	65	45	46	35
	Time	1	5	9	15	1	5	9	6
s1423	#Paths	1323	488	199	137	1339	956	750	638
	#Patterns	306	173	73	50	309	253	171	150
	Time	37	15	18	20	37	44	56	63

VITA

Name: Shayak Lahiri

Address: Department of Computer Science and Engineering
                    Texas A&M University
                    3112 TAMU
                    College Station, TX 77843-3112

Email Address: shayak@cse.tamu.edu

Education: B.Tech, Electronics and Communication Engineering, Motilal
                    Nehru National Institute of Technology, Allahabad, India, 2009
                    MS, Computer Engineering, Texas A&M University, 2011