# RTL Analysis and Modifications for Improving At-speed Test

Kai-Hui Chang
Avery Design Systems, Inc.
Andover, MA, USA
changkh@avery-design.com

Hong-Zu Chou
Avery Design Systems, Inc.
Andover, MA, USA
hzchou@hotmail.com

Igor L. Markov
University of Michigan,
EECS Department
Ann Arbor, MI, USA
imarkov@umich.edu

***Abstract*—At-speed testing is increasingly important at recent technology nodes due to growing uncertainty in chip manufacturing. However, at-speed fault coverage and test-efficacy suffer when tests are not robust. Since Automatic Test Pattern Generation (ATPG) is typically performed at late design stages, fixing robustness problems found during ATPG can be costly. To address this challenge, we propose a methodology that identifies robustness problems at the Register Transfer Level (RTL) and fixes them. Empirically, this improves final at-speed fault coverage and test-efficacy.**

## I. Introduction

The increase in chip manufacturing variations necessitates at-speed testing in addition to the traditional stuck-at fault testing. Otherwise, a chip free of stuck-at failures may still fail to work normally at the intended operating frequency due to delay faults [5], [10]. One commonly used fault model for at-speed testing is the transition delay model. In this model, slow-to-rise and slow-to-fall faults are analyzed for each node. One major limitation of this model is that Automatic Test Pattern Generation (ATPG) tools typically only try to find one transition that sensitizes and propagates a given fault, but such a transition may not sensitize any critical paths through the fault in question. As a result, even if the patterns for transition faults did not detect any problem during testing, critical paths may still experience delay issues. One solution to this problem is to increase the safety margin of operating frequency; however, doing so can significantly reduce parametric yield. Instead, path-delay fault models can be used. Path-delay models target critical paths directly and can detect delay-related problems more accurately. Recently, Small Delay Defect (SDD) testing methods that target transition delay faults in near-critical paths have become popular.

To effectively test path-delay and SDD faults, it is essential to propagate faults along critical paths without the interference from transitions at non-critical paths. The term *robustness* is used to indicate the severity of inference from such side transitions. To improve at-speed ATPG fault coverage and delay effectiveness, it is important to improve test robustness. We observe that most robustness problems are byproducts of the logic in a design. For example, to create a transition at the second bit of a counter, the first bit must toggle. If both bits are in the fan-in cone of a register and the second bit is on the critical path, then the path may not be robustly testable unless the transition at the side-input can be masked. Based on this observation, we propose a new methodology to find and repair robustness problems at the Register Transfer Level (RTL). The main reason for working at the RTL is that it allows the fixes to be optimized by logic and physical design tools, which can aid timing closure [1]. RTL fixes can also be used throughout the design process or even across multiple designs, while gate-level fixes may need to be redone every time the design is resynthesized. Thirdly, a single RTL fix may spawn several gate-level fixes.

978-3-9810801-8-6/DATE12/©2012 EDAA

To improve test robustness at the RTL, we propose an RTL path-delay fault model for predicting robust test coverage. When coverage is low, our novel algorithm can flag robustness problems. It identifies pairs of registers where one always toggles the other. We then use the same RTL path-delay fault model to evaluate the severity of each problem and rank possible fixes. The fixes are implemented with register reuse to improve design robustness. Another contribution in this work is an effective method that can efficiently remove most false paths to reduce false alarms and unnecessary analysis. Our empirical results show that fixing the found robustness problems at the RTL can significantly enhance delay effectiveness and test coverage for ATPG, and our false-path reduction technique can considerably reduce the number of false paths.

The rest of the paper is organized as follows. Section II provides necessary background, and Section III describes our robustness repair methodology. Experimental results are provided in Section IV, and Section V concludes this paper.

## II. Background

In this section, we first review the basics of at-speed testing and then survey current Design for Testability (DFT) solutions at the RTL. We also briefly describe and-inverter graphs.

### A. At-speed Testing

At-speed testing detects delay faults when the circuit is operating at the normal system speed. It is typically performed in four stages: *scan-in*, *launch*, *capture*, and *scan-out*. The purpose of scan-in and scan-out stages is to set/read register values. There are two different test launch types: *Launch-Off-Shift (LOS)* and *Launch-Off-Capture (LOC)* [10]. Due to the limitations of test equipment, LOC is more common. In addition, LOC reduces the chance of over-testing because the tested paths are mostly functional [9]. Therefore, it is

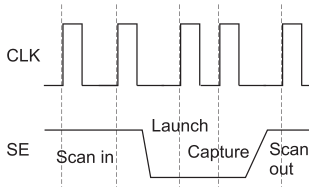


Fig. 1. At-speed testing timing diagram using launch-off-capture method. SE is scan enable and is active high. Launch and capture edges are at-speed.

employed in this work. The timing diagram of a LOC test is shown in Fig. 1 [10].

*Transition fault* and *path-delay fault* are two prevalent fault models used in at-speed testing. A transition fault models large delay defects at a node. A slow-to-rise transition fault represents an unusually slow 0-1 transition, while a slow-to-fall transition fault represents a slow 1-0 transition. The path-delay fault model is used to test distributed delays along a path. If an anomaly causes the signal transition delay along the path to exceed a threshold, the circuit is said to *experience a path-delay fault*. In the path-delay model, the register that creates the value transition is called the *launch register*, and the register to undergo a value change due to the transition is called the *capture register*. Following the terminology in [5], the *side inputs* of the path with respect to the launch register are the registers in the fan-in cone of the capture register excluding the launch register itself.

*Small Delay Defect (SDD)* testing is a variant of the transition-delay model in that it only targets nodes with small timing slacks. To detect small delay problems effectively, the faults need to be sensitized and propagated through the longest timing paths, which is similar to path-delay testing. SDD testing has recently become popular because timing optimization makes numerous paths near-critical, as illustrated by the so-called *timing wall* in the slack histogram. Path-delay testing is unable to cover all necessary paths. SDD testing effectively addresses the path-explosion problem when many near-critical paths pass through the same node. The *delay effectiveness* metric captures the quality of SDD tests [15]. This metric is based on how close the slacks for fault-detection paths came to the minimum slacks. If all faults are detected on their minimum slack paths, the number would be 100%.

*Test robustness* measures whether a fault can be uniquely identified if the captured pattern shows mismatches. With stuck-at-faults, robustness is mainly used for fault diagnosis. With at-speed testing, however, robustness plays a much more important role because, lacking robustness, the fault may not propagate along the critical path, which can considerably reduce delay effectiveness of the tests. To improve test robustness, the side inputs should remain unchanged during the launch cycle so that they do not create unnecessary transitions that affect fault propagation.

### B. Related Work

To improve test robustness, Eggersglüβ [5] proposed a method to generate more robust test patterns using pseudo-Boolean methods. Unlike their work, our focus is to fix the RTL code that causes the robustness problems in the first place. Iwata *et al.* [6] presented non-scan DFT methods for RTL time-expansion models. Another work that fixes DFT problems at the RTL is by Ko *et al.* [7]. However, it improves LOS testability instead of LOC.

### C. And-inverter Graphs

An *And-Inverter Graph (AIG)* is a directed, acyclic graph that represents logic functions in the form of two-input AND gates and inverters. Compared to the most commonly used representations, sum-of-products and *Binary Decision Diagrams (BDDs)*, AIG's multi-level structure is more efficient for manipulating Boolean functions. However, equivalence checking with AIGs is less efficient because AIGs are not canonical. To address such problems, Mishchenko *et al.* proposed a "semi-canonical" representation called *Functionally Reduced AIGs (FRAIGs)* [8] which ensures that each FRAIG node has a unique functionality in a FRAIG structure. In this work, we utilize FRAIGs to reduce the number of false paths.

## III. Our Robustness Improvement Methodology

Fig. 2 illustrates how fixing robustness problems at the RTL can improve the efficacy of gate-level at-speed test. In (a), suppose that there are paths between {*r1, r2, r3*} and *r4*, and we found that paths *r1* to *r4* and *r2* to *r4* are not robustly testable because *r3* cannot be held stable. Now in Fig. 2(b), to test the SDD fault at *node1*, the fault should be sensitized through a path from either *r1* or *r2*. However, since there is a fast path from *r3* to *node1* and *r3* always toggles when *r1* or *r2* toggle, the defect at *node1* can be masked by this fast path, making the fault robustly untestable. Such a situation is common in practice because some registers always toggle simultaneously. For example, to create a transition at the second bit of a counter, the first bit of the counter must also toggle.

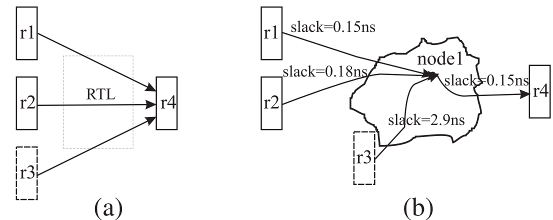


Fig. 2. An illustration of SDD testability problems. (a) At our RTL analysis it is found that path-delay faults from *r1* or *r2* to *r4* are untestable due to *r3* cannot be held stable. (b) At the gate level, if *r3* to *node1* is a short timing path, the small delay fault at *node1* becomes untestable because the transition from *r3* can mask the fault. By fixing the stability problem of *r3* at the RTL, the problem at the gate level can be prevented.

Our solution to the problem is presented in this section, starting with our methodology for improving robustness. We then describe its components in detail: (1) a fault model that aggregates gate-level faults in two faults at the RTL for robust testability estimation; (2) an effective false-path reduction method; (3) a novel algorithm to detect robustness problems; and (4) techniques to repair robustness problems. Finally, we provide insights that we gained during the implementation of our system.

## A. Overall Methodology

A high-level overview of our robustness improvement methodology is shown in Fig. 3. The first step is coverage estimation to determine how severe the robustness problems are. To measure robust testability, we calculate a coverage number that will be described in the next subsection. If the coverage is low, robustness problems should be diagnosed and fixed. The coverage of the repaired design should be estimated again and the repair process can be repeated. This methodology should be applied to each RTL block at its early design phase to avoid problems at the ATPG stage. By focusing on smaller blocks, scalability problems due to the underlying formal algorithms can also be alleviated.

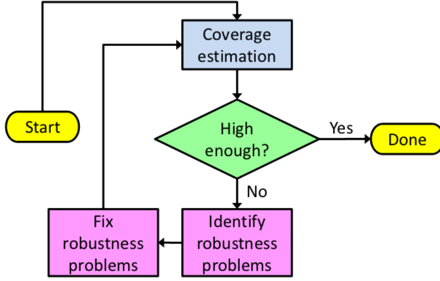


Fig. 3. Our overall robustness improvement methodology.

## B. Robust Test Coverage Estimation

Since no gate-level representation is available at the RTL, it is difficult to precisely estimate either path-delay or SDD fault coverage. In our flow, testability estimation reveals the sensitivity of final coverage to the robustness problems. It is also used to evaluate the impact of robustness fixes. To serve these purposes, we propose a new fault model that only involves the launch and the capture registers, as illustrated in Fig. 4. In our fault model, we test whether slow-to-rise (01) and slow-to-fall (10) transitions at the launch register can affect the capture register. Robustness problems can then be identified when trying to stabilize the values of non-launch registers during the at-speed cycle while inducing the transitions. In this paper, we mark a path as untestable if such transitions cannot be induced. Given that our techniques in Section III-C eliminate most false paths prior to this analysis, the untestable paths are mostly due to the robustness constraints and should be fixed. Note that our analysis is different from traditional fault coverage analysis because we only check whether the fault can be sensitized and propagated robustly. We do not check whether a pattern exists for the fault. However, the former often prevents the existence of the latter.

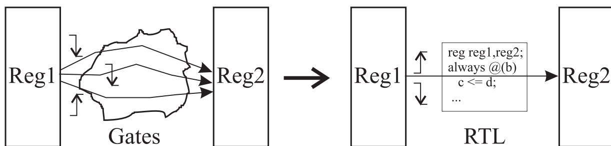


Fig. 4. Our RTL fault model for robust testability analysis. Gate-level faults between Reg1 and Reg2 are aggregated into two faults at the RTL: slow-to-rise and slow-to-fall. For multi-bit variables, we consider each bit separately.

In our fault model, one RTL path contains two faults (slow-to-rise and slow-to-fall) and aggregates all the path-delay and transition faults at the gate level that have the same launch and capture registers. Based on the fault model, Fig. 5 shows our testability estimation flow. The inputs to the flow are an RTL design (*design*) and scannable registers (*sreg*). The outputs of the flow are the testability of each path and the estimated coverage of the design. This flow relies on symbolic simulation and SAT solving, but other Boolean reasoning techniques can also be used.

```
flow TestEst(design, sreg)
 1  sym_traces ← inject_symbol(sreg);
 2  t0 ← collect_sym_traces(registers, sym_traces);
 3  sym_traces ← symbolic_simulate one cycle;
 4  t1 ← collect_sym_traces(registers, sym_traces);
 5  paths ← path_extract(registers, sym_traces);
 6  paths ← false_path_reduction(paths);
 7  sym_traces ← symbolic_simulate one cycle;
 8  t2 ← collect_sym_traces(registers, sym_traces);
 9  foreach path ∈ paths
10    foreach edge ∈ {01, 10};
11      prob_inst ← build_testability_instance(path, t0, t1, t2,
        edge);
12      if (solve(prob_inst) = SAT)
13        path.edge.testable ← true;
14      else
15        path.edge.testable ← false;
16  coverage ← coverage_estimate(paths);
17  return paths, coverage;
```

Fig. 5. Flow for testability estimation.

In TestEst, symbols are injected to scannable registers to represent the scan-in values. Line 3 performs one cycle of symbolic simulation to obtain new register values at launch. Paths between registers are extracted in line 5 using symbolic traces as follows. For each register $rt$, we traverse its symbolic trace to extract the symbols used in the trace and identify the registers where those symbols were originally injected — there is a logic path between each of the registers and $rt$. For example, suppose that symbols injected into $r1$ and $r2$ are in the symbolic trace of $rt$, then there are logic paths between ($r1$, $rt$) and ($r2$, $rt$). We also perform false-path reduction in line 6 (see Section III-C).

After the paths are extracted, we perform symbolic simulation for another cycle in line 7 to model the at-speed testing cycle[1]. At this point, we have the scan-in symbols for each register in $t0$, symbolic traces at launch cycle in $t1$, and symbolic traces at capture cycle in $t2$. Starting from line 9, we check the testability of each path. To achieve this goal, we produce a SAT instance based on the fault model described in Section III-B using the *build_testability_instance* function, and then use a SAT solver to determine whether the instance for the testability problem is satisfiable (lines 11-12). If the instance is satisfiable, then the path is testable. Otherwise, the path is untestable. We then estimate path-delay fault coverage on line 16 and then return the results.

[1]If the path being tested is a multi-cycle path, more cycles of symbolic simulation should be performed here.

Function *build_testability_instance* is shown in Fig. 6. Suppose that the launch register in *path* is *rl*, the capture register is *rc*, and $r1..rn$ are in the fan-in cone of *rc*. We use subscript *sn* to represent the symbolic trace collected in $tn$, $n \in \{0,1,2\}$. In the function, lines 2-5 initiate the value transition at the launch register, which can be either 01 or 10 (slow-to-rise or fall). Line 6 ensures that the capture register can detect the signal change, and lines 7-8 make sure all other registers do not toggle during the at-speed cycle.

```
function build_testability_instance(path,t0,t1,t2,edge)
1   (rl, rc) ← extract_reg(path);
2   if (edge is 01)
3     instance ← ¬rl_s0 & rl_s1;
4   else
5     instance ← rl_s0 & ¬rl_s1;
6   instance ← instance & (rc_s1 ≠ rc_s2);
7   foreach r ∈ {r1..rn}, r ≠ rl
8     instance ← instance & (r_s0 = r_s1);
9   return instance;
```

Fig. 6. Function *build_testability_instance* that builds a SAT instance for checking robust testability. Symbolic traces are saved in $t0$, $t1$ and $t2$. Their retrieval is omitted in the figure.

### C. False-path Reduction

False paths are untestable and excluding them improves the accuracy of analysis [2]. At the RTL our goal is to remove paths from fan-in registers of a capture register that are not in its functional support. However, finding all false paths can be time-consuming [4]. Since we want to eliminate as many false paths as possible but not necessarily all of them, we propose a new structural FRAIG-based technique.

1) Represent the Boolean function by an AIG.
2) Build a FRAIG from the AIG by SAT-based node merging while distinguishing nodes using simulation.
3) Traverse the FRAIG from its output to identify all the primary inputs that still connect to the output.

FRAIGs work well in our application because their compact size reduces the likelihood of false paths. Our experience shows that FRAIGing can eliminate most false paths and scales well in practice.

### D. Robustness Analysis

Most robustness problems are caused by controllability issues involving three registers: the launch register *rl*, the capture register *rc*, and a side-input register *r* in the fan-in cone of *rc*. More specifically, a problem arises when initiating a transition at *rl* and observing the resulting transition at *rc* requires register *r* to toggle unconditionally. When this happens and $r1$ is on the critical path, the fault may not be sensitized and propagated through the critical path due to the transition from *r*. Based on this observation, we propose a new algorithm to directly detect such configurations. The algorithm is based on flow *TestEst* in Fig. 5 with two changes. First, in line 11, the call to function *build_testability_instance* should be replaced with a call to the new function, *check_robustness_problem*, shown in Fig 7. Second, lines 12-17 are not necessary because the problems are directly reported by the new function.

```
function check_robustness_problem(path,t0,t1,t2,edge)
1   (rl, rc) ← extract_reg(path);
2   if (edge is 01)
3     cond ← ¬rl_s0 & rl_s1;
4   else
5     cond ← rl_s0 & ¬rl_s1;
6   cond ← cond & (rc_s1 ≠ rc_s2);
7   if (solve(cond)) = UNSAT)
8     return;
9   foreach r ∈ {r1..rn}, r ≠ rl
10    instance ← ¬(cond ⇒ (r_s0 ≠ r_s1));
11    if (solve(instance)) = UNSAT)
12      report robustness problem (r);
```

Fig. 7. Function *check_robustness_problem* that checks robustness issues. Symbolic traces are saved in $t0$, $t1$ and $t2$. Their retrieval is omitted.

In Fig. 7, *cond* represents the Boolean condition for both the launch and capture registers to toggle. If this condition is judged impossible (line 7), the path is false, and no further checking is necessary. In lines 9-12 we iterate through each register (*r*) in the fan-in cone of *rc* and check if *cond* implies that *r* must also toggle. If the solver proves that the implication always holds, then it may not be possible to sensitize and propagate faults from the launch register *rl* to the capture register *rc* because of the transition in *r*. If the critical paths happen to involve *rl* and *rc*, then the faults on those paths will be potentially untestable. We report register *r* as a potential cause of robustness problems in line 12.

In our use model, we count the number of times that a register *r* causes problems and then rank the fixes accordingly — registers ranked high should be repaired first because they affect more paths. To reduce runtime, one can also remove the transition at the capture register from the problem formulation and check robustness issues only at the launch cycle. Even though this formulation only considers fault sensitization, it provides almost the same information as the original formulation. The reason is that robustness is mostly associated with fault sensitization, making the transition at the capture cycle less relevant. In this formulation, symbolic simulation only needs to be performed for one cycle instead of two, which can considerably reduce analysis time.

Given a path, the algorithm checks stability between the launch register and each side input, which can be time-consuming. We observe that sometimes only a few side inputs cause robustness problems. To accelerate the analysis, we perform binary search — when only a few side inputs are problematic, runtime can be dramatically improved. However, runtime may increase otherwise. In practice, one can determine which mode is faster by sampling a small percentage of paths before performing full analysis.

### E. Repairing Robustness Problems

To stabilize a register, we insert one multiplexer (MUX) and two scannable registers, *mode_select* and *new_value*. The MUX then selects between the original output of the register, *reg*, and the new launch value register, *new_value*, as follows:

original output = *mode_select* ? *new_value* : *reg*

Register *mode_select* is set to 1 only in test mode. This allows us to change the launch value of a register. To reduce the amount of inserted logic, existing scannable registers can be reused. To this end, we identify the side inputs of *reg* and select a register which is not part of those inputs as the *new_value* register. To insert fewer *mode_select* registers, we analyze the side inputs of each fix and group the fixes whose side inputs are independent — the *mode_select* register can be shared by the fixes in the same group. If the newly-introduced timing path is judged critical, the inserted MUX can be retimed to the input of *reg*. Our MUX insertion can be compared to *partial enhanced-scan* in [3], but our selection of registers for repair is quite different.

## IV. Experimental Results

We applied our robustness improvement methodology to several designs to estimate their robust test coverage and measure their robustness improvement after repair. The characteristics of the designs are summarized in Table I. In the table, "#Cells" is the number of standard cells in the design mapped with the TSMC 0.13$\mu$m library. DLX is from Michigan [12], and the rest of the designs are from OpenCores [14]. We implemented our methodology using a commercial symbolic simulator [11] and the ABC package from Berkeley [13]. Our machine was a Dell PowerEdge 2900 (2GHz Quad-Core Xeon, 48 Gbytes main memory) running Linux. Three mainstream commercial tools, labeled SY, TA and PG (license agreements prevent us from naming them), were used for synthesis, critical path extraction and path-delay/SDD fault coverage comparison, respectively. Note that PG was configured so that the path-delay faults were robustly testable.

TABLE I
Characteristics of benchmarks.

| Design | #Cells | #Reg. | Description |
|---|---|---|---|
| DLX | 6075 | 1543 | MIPS-lite 5-stage CPU |
| CPU8080 | 1420 | 242 | 8080 compatible CPU |
| TV80 | 1010 | 358 | Z-80 compatible CPU |
| AES_core | 10657 | 530 | AES encoder |
| USB_funct | 7688 | 1745 | USB function IP core |
| USB_phy | 157 | 98 | USB physical layer |

### A. Robust Test Coverage Estimation

The results of our testability estimation are summarized in Table II. To obtain fault coverage in a traditional flow, we first synthesized the design with SY, extracted up to 200 critical paths using TA, and then used PG to calculate fault coverage of the paths. In addition, we used PG to calculate SDD fault coverage and delay effectiveness. The results of our testability estimation methodology are reported under "our methodology", and the results of the traditional flow are reported under "traditional flow". We report our per-path runtime based on the total runtime divided by the number of paths before reduction. This represents the estimated time to check one path that is structurally connected in the RTL code. Memory usage of our methodology was under 2G for the benchmarks.

Coverage comparison shows a difference between estimated and final coverage after ATPG. This is not surprising because the RTL code we analyze lacks most of the information required to obtain accurate coverage numbers. In addition, our formulation only considers untestability caused by robustness problems. Nevertheless, our coverage estimation for path-delay faults is accurate to 15% for most benchmarks. One exception is DLX, where our coverage was 21.94% and the final coverage was 43.00%. We analyzed the difference and found that SY decomposed the register array into individual registers, while our flow modeled the array as a memory. After manually changing the design to match the output of SY, our estimated coverage increased to 43.92% — an exception that proves the rule. As expected, this considerably increased runtime due to the extra overhead to handle the large number of registers. The comparison between "TA paths" under the traditional and our flow shows that our coverage was typically higher. This is because the fault in our model is testable if any of the aggregated gate-level paths are testable, making our model optimistic. This observation can also explain the large difference between the estimated coverage and the final coverage for AES_core, where our estimation was 92.02% and the ATPG coverage was 67.50%. The special structure of encoders creates numerous gate-level paths between registers and makes more RTL paths testable because each RTL path requires only a single testable gate-level path to be testable. From the results, we also observe that the SDD fault coverage is much higher than path-delay fault coverage. This is also expected because many of the SDD faults are on simple short paths that can be detected easily. However, the results show that there is correlation between our predicted coverage and the SDD fault coverage. For the cases where the predicted coverage were low, such as DLX, CPU8080 and TV80, the SDD coverage was also lower compared with other cases. This result shows that our coverage estimation can also be useful for predicting SDD fault coverage problems due to robustness.

To evaluate the effectiveness of our false-path reduction method, we report the number of extracted paths with and without reduction under "#Paths". Without reduction, the paths were extracted based on the logic structure of the design. From the results, we found that the use of FRAIG significantly reduced the number of false paths. We also found that the time used to build FRAIG for false-path reduction was well-spent because it reduced the number of paths that need to be checked, thus reducing the total SAT solving time. SAT solving was further accelerated because SAT instances produced from FRAIG structures were more compact. As a result, false-path reduction paid off by considerably reducing overall runtime.

### B. Robustness Repair

For a given netlist, critical paths are often determined by physical layout. Therefore, we evaluate our robustness-repair methodology using SDD fault coverage and the delay effectiveness metric [15]. We chose CPU8080 and TV80 for repair because they had the lowest predicted coverage.

TABLE II
ROBUST TEST COVERAGE ESTIMATION. "SDD (COV/EFF)" IS SMALL DELAY DEFECT COVERAGE AND DELAY EFFECTIVENESS. "TA PATH COVERAGE" IS CALCULATED USING CRITICAL PATHS EXTRACTED BY TA. RUNTIME UNDER "OUR METHODOLOGY" IS PER-PATH RUNTIME.

| Design | Traditional flow coverage | | Our methodology | | | | |
|---|---|---|---|---|---|---|---|
| | | | #Paths | | Predicted coverage | Coverage (TA path) | Runtime (pp, sec) |
| | Path | SDD(cov/eff) | w/o reduction | w/ reduction | | | |
| DLX | **43.00%** | **96.76%/90.56%** | 99230 | 70324 | **21.94%** | 46.49% | **0.1107** |
| CPU8080 | **0.00%** | **72.31%/68.31%** | 17028 | 7420 | **11.68%** | 11.94% | **0.0099** |
| TV80 | **0.00%** | **89.01%/83.43%** | 24846 | 15684 | **9.02%** | 29.03% | **0.0063** |
| AES_core | **67.50%** | **96.54%/99.10%** | 49800 | 11710 | **92.02%** | 100% | **0.0170** |
| USB_funct | **45.09%** | **88.38%/96.16%** | 197386 | 49468 | **50.85%** | 71.90% | **0.0248** |
| USB_phy | **40.57%** | **86.62%95.12%** | 1196 | 928 | **34.96%** | 67.07% | **0.0008** |

For CPU8080, test coverage was 72.31% with delay effectiveness 68.31%. We first applied the algorithm shown in Fig. 7. It was found that 35 variables could cause robustness problems, and runtime was 48m25s. We then applied the modified algorithm which only considers the robustness at the launch cycle. The same 35 variables were found but runtime was reduced to 2m37s. The only difference was the minor reordering in the ranking of the variables. Due to its superior performance, the modified algorithm is used from now on. An analysis based on the number of registers affecting each path shows that 90% of the robustness problems involve $\leq 3$ registers, making them easy to fix. We repaired the highest-ranking word-level variable, *state[6:0]*, at the RTL, and the area overhead was 1.1%. The test coverage after repair was 92.87% with delay effectiveness 88.62%, improved from 72.31% and 68.31%, respectively. This result shows that by performing robustness analysis and repair at the RTL, the SDD fault coverage can be improved effectively at the ATPG stage. In addition, ATPG runtime decreased from 1m6s before repair to 25s after repair. This shows another benefit of applying DFT analysis and repair at the RTL — by fixing problems in advance, ATPG runtime can be reduced by not looking for impossible-to-generate patterns.

To evaluate the effectiveness of our repair ranking method, we performed another experiment by repairing the lowest-ranking variable, *waddrhold[15:0]*. The highest-ranking variable (*state[5:0]*) affected 3875 paths, while *waddrhold* only affected 136 paths. The test coverage after repair was 72.27% with delay effectiveness 68.17%, showing little change in both numbers. This result shows that our ranking method is effective in pointing out problems that should be fixed first.

For TV80, we found that 195 out of 242 registers can cause robustness problems. Runtime was 11m14s for the analysis. We repaired 25 bit-level registers, and the area overhead was 5.0%. Test coverage improved from 89.01% to 90.40%, and delay effectiveness improved from 83.43% to 83.72%. The improvement is not as impressive as for CPU8080 because 352 out of the total 368 paths involve more than 10 robustness problems, making the problems more difficult to repair. Considering such grouping during repair ranking is our ongoing work.

Runtime of our robustness analysis increases with the number of paths to analyze. The above results suggest that repairing problems affecting more paths is more effective. Therefore, a sampling scheme that only analyzes a percentage of all paths can be used to improve the scalability of our methods. Since problems affecting more paths will be sampled more often, important problems will not be missed.

## V. CONCLUSION

Due to the increase in process variations, at-speed testing is becoming more important. Robustness is one important factor that impacts delay effectiveness of the tests. However, repairing robustness problems at the ATPG stage can be costly; therefore, it is desirable to have a DFT solution to find an fix robustness problems at the RTL. This work proposed a methodology to achieve this goal. To support this methodology we designed an RTL fault model consistent with the gate-level path-delay and SDD fault models for robust test coverage estimation, developed an effective and efficient false-path reduction method, devised a robustness analysis technique, and illustrated how to repair the robustness issues. Empirically, these techniques are informative at early design stages and effective in improving test robustness.

**Acknowledgment:** The authors want to thank Zahi Abuhamdeh (SiliconDFx) for motivating this work.

## REFERENCES


[1] E. Alpaslan, Y. Huang, X. Lin, W.-T. Cheng, J. Dworak, "Reducing Scan Shift Power at RTL", *VTS'08*, pp.139-146.

[2] D. Blaauw, R. Panda, A. Das, "Removing user specified false paths from timing graphs", *DAC'00*, pp. 270-273.

[3] K.-T. Cheng, S. Devadas and K. Keutzer, "A Partial Enhanced-Scan Approach to Robust Delay-Fault Test Generation for Sequential Circuits", *ITC'91*, pp. 403-410.

[4] O. Coudert, "An Efficient Algorithm to Verify Generalized False Paths", *DAC'10*, pp. 188-193.

[5] S. Eggersglüβ and R. Drechsler, "As-Robust-As-Possible Test Generation in the Presence of Small Delay Defects using Pseudo-Boolean Optimization", *DATE'11*, pp. 1291-1296.

[6] H. Iwata, T. Yoneda, and H. Fujiwara, "A DFT Method for Time Expansion Model at Register Transfer Level", *DAC'07*, pp. 682-687.

[7] H. F. Ko and N. Nicolici, "Functional Scan Chain Design at RTL for Skewed-load Delay Fault Testing", *ATS'04*, pp. 454-459.

[8] A. Mishchenko, S. Chatterjee, R. Jiang, R. Brayton, "FRAIGs: A Unifying Representation for Logic Synthesis and Verification", ERL Technical Report, EECS Dept., U. C. Berkeley, March 2005.

[9] J. Rearick, "Too Much Delay Fault Coverage is a Bad Thing", *ITC'01*, pp. 624-633.

[10] B. Swanson and M. Lange, "At-Speed Testing Made Easy", *EE Times*, Jun. 3, 2004.

[11] Avery Design Systems Inc., http://www.avery-design.com

[12] Bug UnderGround, University of Michigan, http://bug.eecs.umich.edu

[13] Berkeley Logic Synthesis and Verification Group, ABC: A System for Sequential Synthesis and Verification, http://www.eecs.berkeley.edu/~alanmi/abc/abc.htm

[14] OpenCores, http://www.opencores.org

[15] "TetraMAX ATPG User Guide", Synopsys, Dec. 2008.