VLSI Testing Final Project Report

R07943091 錢柏均 R06943085 黃韋智
R07943158 Divya Jain R07943108 朱宇融

1. Introduction

In this project, we need to modify the atpg code provided by TA to support the test pattern generation for transition delay fault. The generation of patterns need to follow the launch-on shift technique to test the fault. We can also use the static and dynamic compression technique to reduce the number of test patterns. Our goal is to generate the test patterns with high quality (high fault coverage), low runtime and small number of test patterns.

2. Previous work

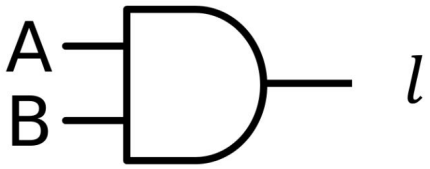
In the reference paper ”Compact test generation with an Influence input measure for launch-on-capture transition fault testing”. They proposed a method similar to SCOAP to help estimate the difficulty of a transition delay fault to be detected. Three key concepts are:

I. Control Reachability (RC_i)
Minimum set of PI assignment to set a wire’s value to i(0/1). Starting from PI, whose RC_1 is itself and RC_0 is its complements. The wires in the circuit will be visited in topological order, and the smallest set of PI assignment will be selected to make the wire’s value to be 0 or 1, e.g. for the AND gate in the figure below:

$$RC_1(l) = RC_1(A) \cup RC_1(B). \qquad RC_0(l) = \begin{cases} RC_0(A), & \text{if} |RC_0(A)| \leq |RC_0(B)| \\ RC_0(B), & \text{if} |RC_0(A)| > |RC_0(B)|. \end{cases}$$

II. Observation Reachability (RO)
Minimum set of PI assignment to observe a wire’s value. Starting from PO, whose RO is an empty set. Each wire will be visited in reverse topological order, RO will be the union of PI assignments to set the side inputs of the gate to non-controlling value and the RO og the gate’s ouput wire, e.g. for the AND gate in the figure below:

$$RO(A) = RO(l) \cup RC_1(B).$$



III. Fault Detectability (det)

Minimum set of PI assignment to detect a fault. After we compute the RC and RO of each wire, we can compute the fault detectability by the formula shown below:

1st timeframe 2nd timeframe

$$\det(l/i) = RC_j(l) \cup RC_{\bar{j}}(l') \cup RO(l')$$

Since we are interested in transition delay faults, there will be input assignments of 2 timeframes. And we only need to propagate the fault in the 2nd timeframe, thus we don't need to add the RO constraint int the 1st timeframe. e.g. for a STR(0→1) fault at wire l:

$$det(l\ STR) = RC_0(l) \cup RC_1(l') \cup RO(l')$$

Complexity: Let n = number of PIs, m = circuit size. Time: O(m). Space: O(n*m).

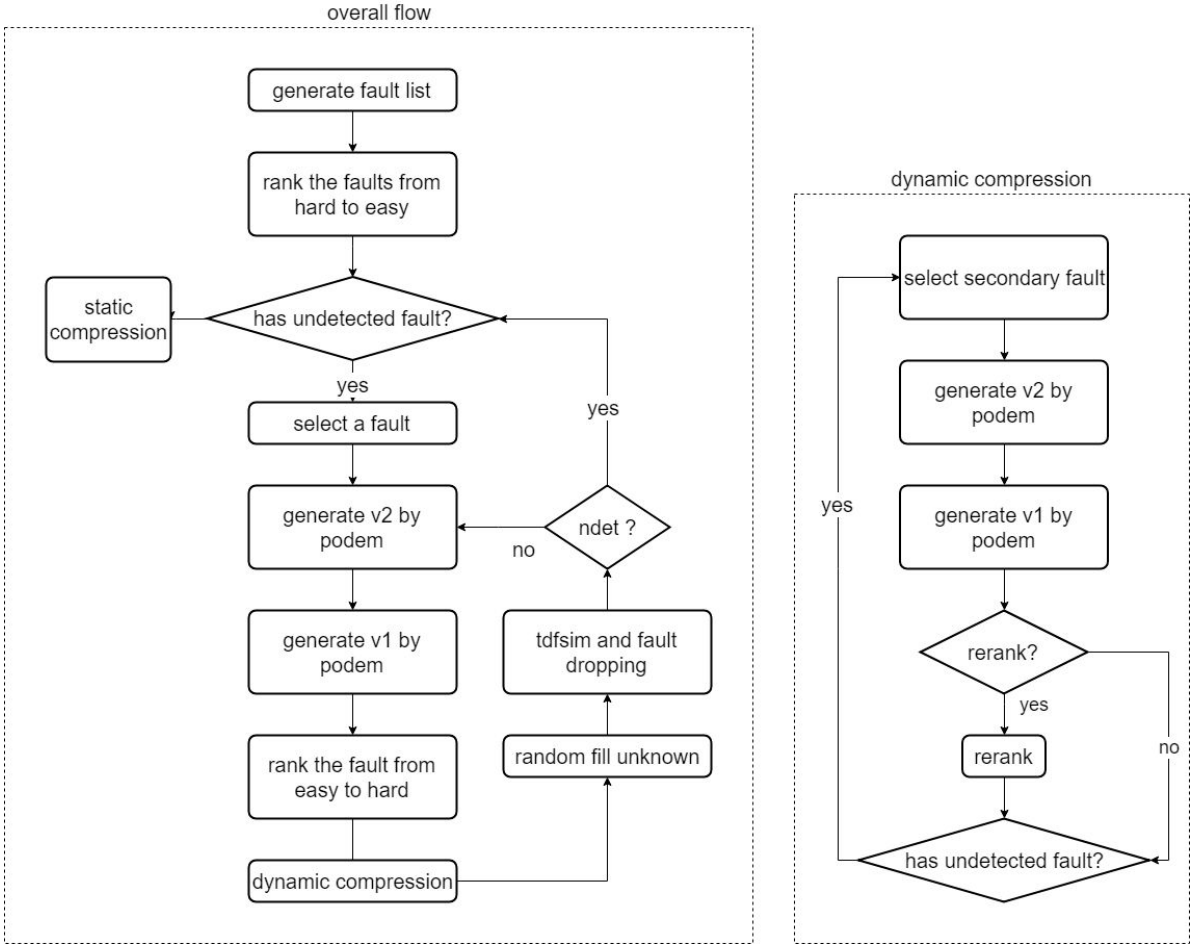
3. Our Method

I. Overall Flow

The figure below is the overall flow chart of our implementation. For the selection of fault, we rank the fault, and choose the fault that is difficult to test first. Since fault that is easier to test is likely to be test by the pattern of fault that is difficult to test, we believe this will make the number of test pattern smaller. This way, it may take less time to generate all the test patterns.

We have modified the podem algorithm to support input constraints and storing the decisions. In the pattern generation of v1, we use the shifted v2 as the input constraints, and we invert the fault from STR (STF) to STF (STR) because the desired wire value for v1 and v2 are inverted. Also, since we only need v1 to activate the wire and do not need to propagate to PO, we add a flag in the podem algorithm to support the idea. If it fails to generate a pattern in such constraints, the we rerun the podem for v2 with the storing decisions, so that it will not generate the same pattern again.

After generating a pattern, if there is unknown in it, we randomly fill the unknown with 0 or 1. This may detect more faults.



II. Fault Ranking

- Modification of detectability
  The TDF testing technique we focused on is LOS, which is different from the LOC in the reference. Thus we add additional LOS constraints on the fault detectability the proposed. That is, we shift the PI assignment in the 1st(2nd) timeframe and set it as the constraints of the 2nd(1st) timeframe.

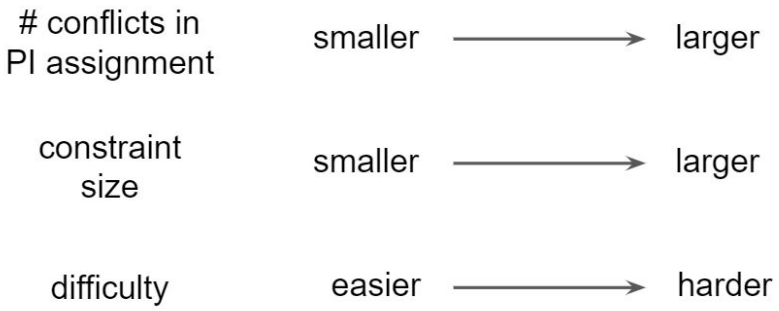
1st timeframe 2nd timeframe

$$\det(l/i) = RC_j(l) \cup RC_{\bar{j}}(l') \cup RO(l') \cup \{LOS\ constraint\}$$

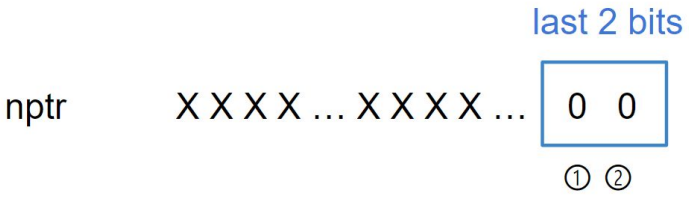
- Ranking criteria
  We will set up the constraint for each fault as followed:

  constraint = current assignment ∪ detectability

To decide which fault is easier to be detected, we will first sort the faults by the number of conflicts in their constraint set. A conflict means that the same PI is assigned to both 0 and 1 in the same timeframe. If the number of conflicts is tied, then we will sort the faults by the size of their constraint set, that is, the number of PIs has to be assigned. The smaller number of conflicts and constraint size means the fault should be easier to be detected.



- Implementation details
  Each PI is represented by its node pointer. Inspired by the implementation of BDD and AIG node, we know that the last 2 bits will be both 0s in a legal pointer, so we can use them to store information without using extra memory. The last bit is used to distinguish which value the PI is assigned, and the second-last bit is used to separate the 2 timeframes. As shown in the figure below:



① 1st timeframe : 0, 2nd timeframe : 1

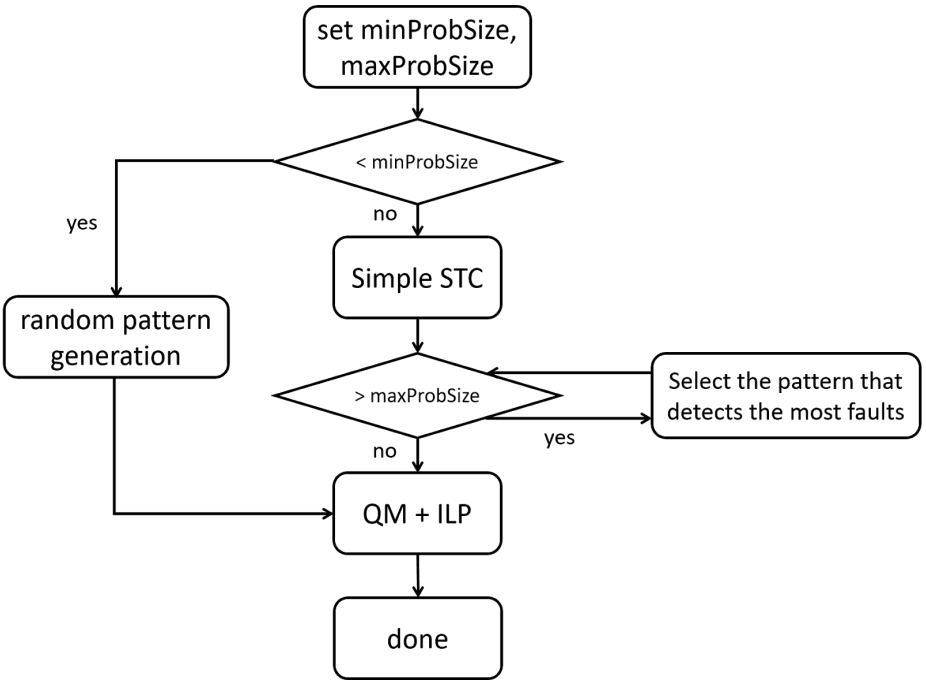
② assign 1 : 0, assign 0 : 1

III. Dynamic Test Compression

In dynamic compression, we sort the fault from the easier fault to harder fault. We believe it can compress more fault by this order. However, if the number of fault to be compressed is large, it takes a lot of time to sort the faults. To reduce the runtime, we sort only a small fraction of fault. To compress the secondary fault, the pattern for primary fault is used as the input constraints of the podem algorithm. We iterate the procedure until there is no fault can be compressed into the pattern.

IV. Static Test Compression

We implemented 3 kinds of STC algorithms, the overall flow is shown in the figure below.



- Simple STC (random/reverse order simulation)
  During fault simulation, we observe which patterns are necessary. So we keep only those patterns and remove the redundant patterns. In order to achieve test compaction of generated test set, a fault simulation in reverse order is performed at final stage of ATG process.

- Tseng-Siewiorek (TS) Algorithm
  We also implement Tseng-Siewiorek Algorithm, but the unknown of test patterns has randomly been filled with 0 or 1 for improving fault coverage, so TS Algorithm become unnecessary. At last we don't use it for compression.

- Quine-McCluskey (QM) + ILP
  We set up upper and lower bounds for the problem size, which are determined by the number of faults and patterns. If the original problem size is smaller than the lower bound, the we will randomly generate more patterns. In this case, we can have more pattern candidates and there are chances that we can improve our fault coverage. If the problem size exceeds the upper bound, then we will

first perform simulation of each pattern and record the number of faults detected. The patterns that detected the most faults are selected until the remaining pattern candidates forms a problem whose size is within the upper bound. When the problem size is within an acceptable range, we will build a fault dictionary from the pattern candidates. Note that for larger circuits, the fault dictionary will be "partial" since we have already select some patterns by heuristics. The fault dictionary is stored as a 2D bit vector (binary matrix, each entry takes only a bit) $[b_{ij}]_{n\times m}$ (n: number of detectable faults, m: number of pattern candidates), and new variables v_j are introduced. Their meanings are shown below.

$$\begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1m} \\ b_{21} & b_{22} & & b_{2m} \\ \vdots & & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nm} \end{bmatrix} \quad b_{ij} = \begin{cases} 1, if\ fault\ i\ can\ be\ detected\ by\ pattern\ j \\ 0, \quad otherwise \end{cases}$$

$$v_j = \begin{cases} 1, if\ pattern\ j\ is\ selected \\ 0, \quad otherwise \end{cases}$$

We will then set up the constraints

$$for\ i = 1 \sim n, \qquad \sum_{j=1}^{m} b_{ij} \cdot v_j \geq ndet - d_i$$

where d_i is the number of times the fault i has be detected by the already selected patterns. The objective is to minimize the sum of v_j, that is, the number of patterns selected from the candidates.

$$min: \sum_{j=1}^{m} v_j$$

After the constraints and objective are correctly set up, we will call an external ILP solver(lp_solve 5.5) to solve this problem. An additional timeout limit will be added since our program has to terminate within 10 minutes. An optimal or sub-optimal solution will be returned if possible and we will then compress the patterns based on the results.

V. N-Detect

To generate test patterns for n-detect, when we select a primary fault, we iteratively generate different patterns until the fault is n-detected or failing to generate a pattern. After all the fault are selected, if a fault can be test by

some pattern, we store a pattern that detect the fault when generating the pattern, we duplicate the pattern to ensure that a fault can be detect n times. This way, the fault coverage will not decrease with respect to n detect.

3. Experimental Results

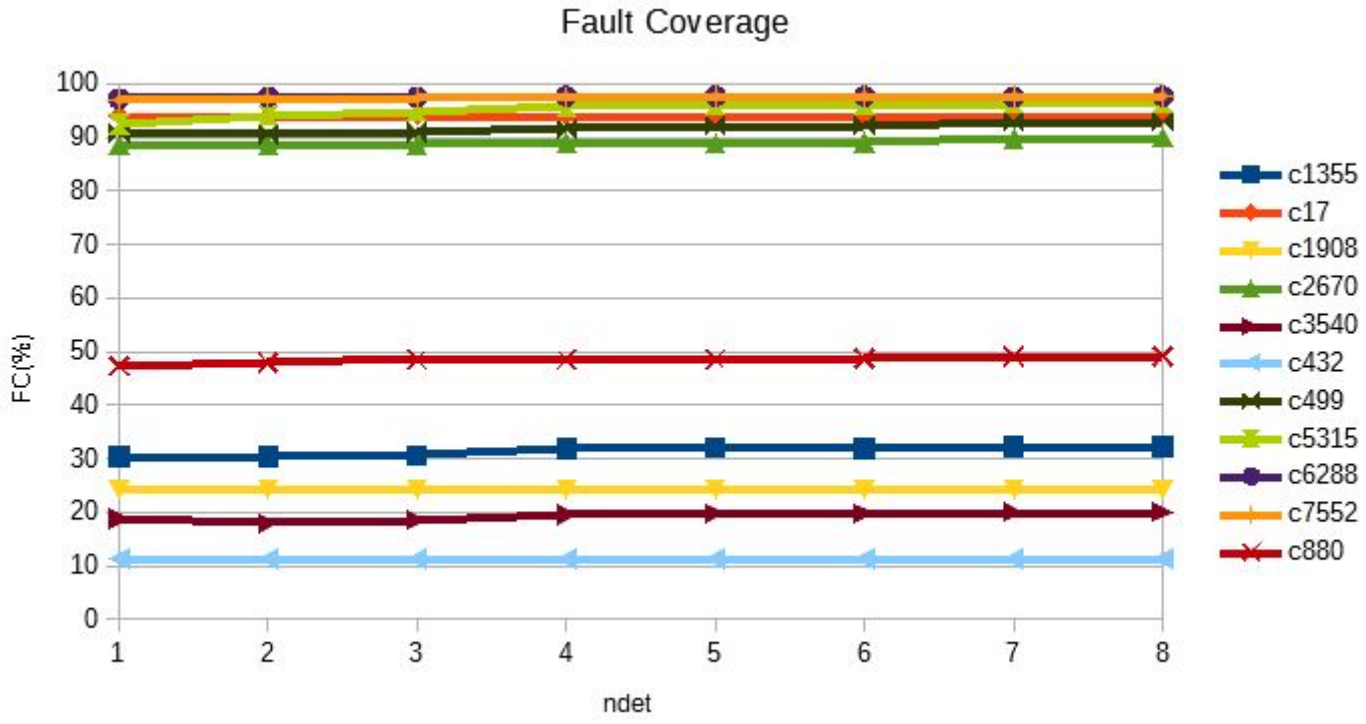
We run the experiments on a server with Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz and 125G RAM. The results of 1 and 8 detect are shown below. The length reduction in the last column is defined by $\frac{\textit{test length without compression} - \textit{test length with compression}}{\textit{test length without compression}}$.

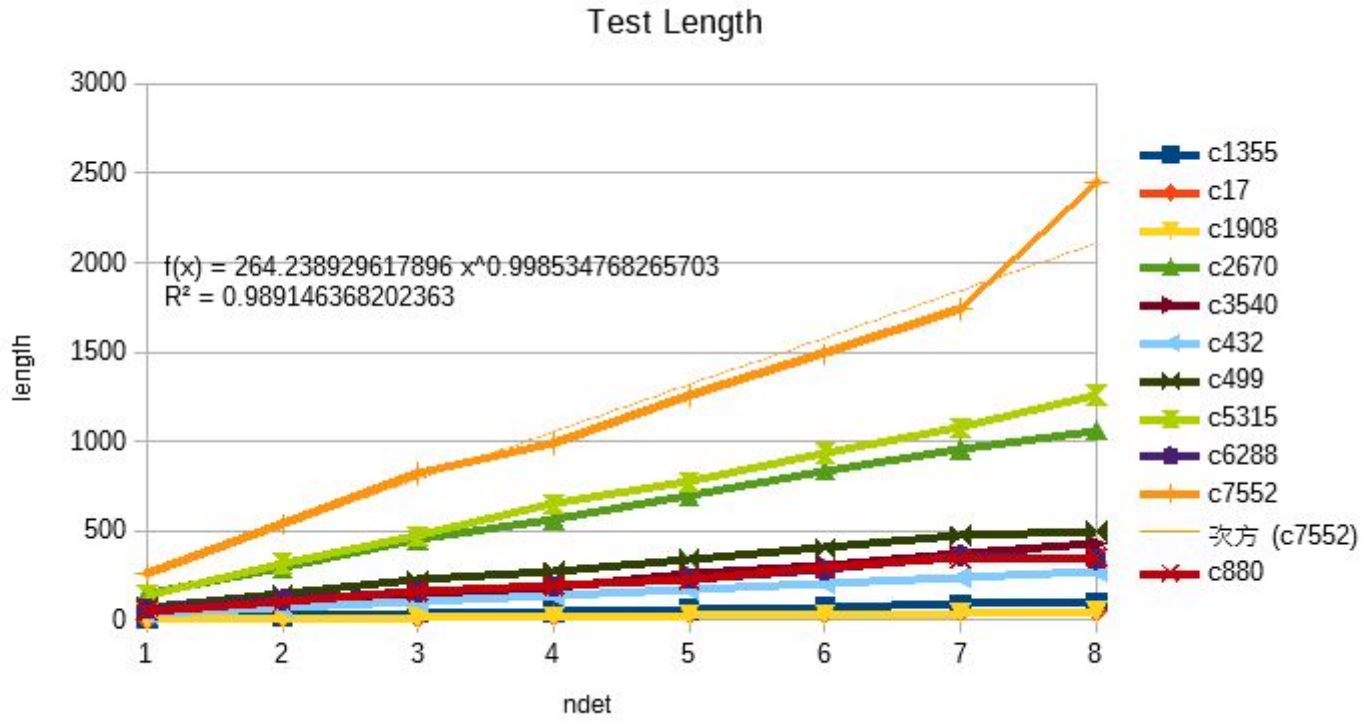
ndet 1	without compression			with compression			
circuit	FC(%)	length	time(s)	FC(%)	length	time(s)	length reduction(%)
c17	88.235	9	0	94.117	6	0	33.33%
c432	11.441	34	0	11.441	34	0	0.00%
c499	90.502	89	1	90.836	76	1	14.61%
c880	45.484	59	0	47.338	53	2	10.17%
c1355	30.41	17	2	30.41	13	2	23.53%
c1908	24.319	10	2	24.319	5	2	50.00%
c2670	88.266	193	6	88.588	155	34	19.69%
c3540	16.725	45	14	18.824	49	17	-8.89%
c5315	92.619	182	8	92.411	143	27	21.43%
c6288	97.208	112	47	97.185	66	49	41.07%
c7552	95.877	422	24	96.89	266	105	36.97%
average	61.92	106.55	9.45	62.94	78.73	21.73	21.99%

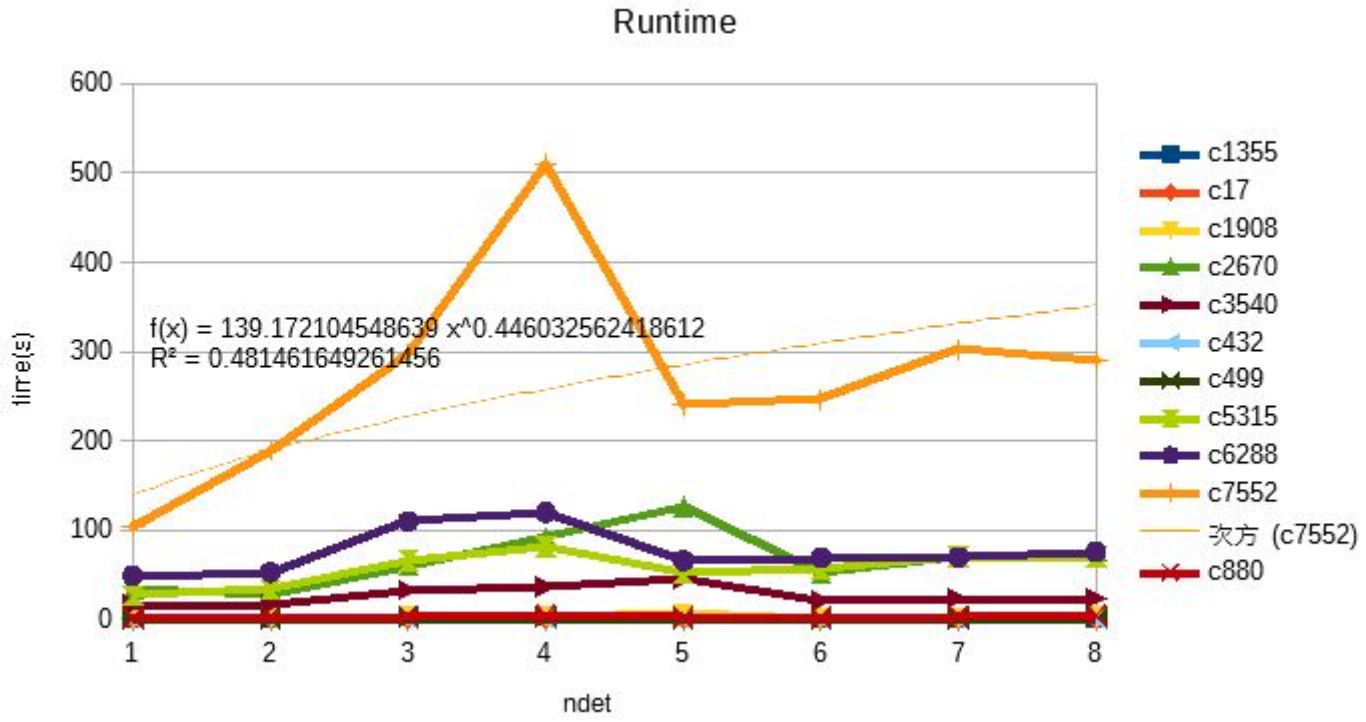
ndet 8	without compression	with compression	

circuit	FC(%)	length	time(s)	FC(%)	length	time(s)	length reduction (%)
c17	88.235	90	0	94.117	48	0	46.67%
c432	11.441	499	0	11.441	272	0	45.49%
c499	93.012	772	1	92.97	500	2	35.23%
c880	49.287	716	0	49.144	352	5	50.84%
c1355	33.125	594	2	32.355	101	2	83.00%
c1908	24.345	838	2	24.345	48	3	94.27%
c2670	89.601	1684	7	89.907	1064	76	36.82%
c3540	19.81	1008	14	20.113	426	24	57.74%
c5315	96.76	1829	10	96.383	1259	70	31.16%
c6288	97.651	896	53	97.651	343	76	61.72%
c7552	97.065	3637	33	97.738	1961	394	53.958%
average	63.67	1142.09	11.09	64.20	624.09	49.91	52.32%

We run different n detect to see how the fault coverage, test length and runtime change with respect to n detect.


Fault Coverage
FC(%)
ndet
c1355
c17
c1908
c2670
c3540
c432
c499
c5315
c6288
c7552
c880


Test Length
length
ndet
f(x) = 264.238929617896 x^0.998534768265703
R² = 0.989146368202363
c1355
c17
c1908
c2670
c3540
c432
c499
c5315
c6288
c7552
次方 (c7552)
c880



From the results, the fault coverage does not change a lot with n detect. The reason is that we duplicate test patterns to ensure the fault coverage is at least the same as 1 detect. The test length and runtime increases linearly with n detect from the figure. There is a glitch in the runtime of circuit c7552, this may be caused by the runtime limitation of QM which makes sure that the runtime do not exceed 10 minutes.

We then run the experiments to see how effective the compression is. The experiments are run with 8 detect.

	simple static with dynamic	dynamic	simple static without dynamic
ckt	length reduction	length reduction	length reduction
c17	0.00%	0.00%	0.00%
c432	0.00%	0.00%	0.00%
c499	0.00%	0.65%	0.00%
c880	0.00%	21.65%	0.00%
c1355	0.00%	20.03%	0.00%

c1908	93.68%	0.00%	93.68%
c2670	20.96%	3.38%	20.43%
c3540	38.38%	14.19%	46.43%
c5315	24.72%	-10.83%	14.38%
c6288	48.21%	-12.05%	42.97%
c7552	20.05%	15.67%	26.07%

Since we randomly add test patterns in the stage of QM, this makes it difficult to check how effective QM is, we do not run the test. From the statistics, it shows the static compression is more effective than dynamic compression in our implementation.

Finally, we run the experiment to see how effective the selecting difficult fault first is, the experiments are run with 8 detect and without compression. The fault coverage difference, length reduction and runtime reduction are defined by $fault\ coverage\ of\ difficult\ first - fault\ coverage\ of\ easy\ first$, $\frac{test\ length\ of\ easy\ first - test\ length\ of\ difficult\ first}{test\ length\ of\ easy\ first}$ and $\frac{runtime\ of\ easy\ first - runtime\ of\ difficult\ first}{runtime\ of\ easy\ first}$, respectively.

ckt	fault coverage difference(%)	length reduction(%)	runtime reduction
c17	-5.88	15.89%	NaN
c432	0	-54.49%	NaN
c499	0.08	14.03%	50.00%
c880	-0.48	0.83%	100.00%
c1355	0.81	10.00%	0.00%
c1908	0	-490.14%	0.00%
c2670	-0.92	0.47%	85.42%
c3540	-1.16	21.62%	72.55%
c5315	-0.04	14.45%	86.30%
c6288	-0.006	-101.35%	-120.83%

c7552	-0.77	-19.76%	82.99%

From the results, in many cases, the fault coverage does not change a lot, but the runtime reduction is significant. However, it is not clear whether the test length is better. So the assumption we made, which says that the test length will be small when selecting difficult fault first, is not correct.

4. Discussion

We have successfully implemented the atpg for transition delay fault, and compress the test patterns. The experimental results are run to check if the results meet our expectation. Though it seems we have made wrong assumption on the test length with respect to fault difficulty, this may be caused by how we decide which fault is difficult. We may need to change our mind to find which fault is better for selecting first rather than how difficult it is. However, by selecting difficult fault first, we can reduce the runtime of test pattern generation in many cases. This can be that there is more fault being dropped by the pattern of difficult fault, which reduce the candidate of dynamic compression, so less runtime.

5. Team Member Contributions

- 錢柏均
  fault ranking, static test compression(simple, QM), part of ATPG flow, testsuite(test the performance of our program and keep logs automatically), final report(the implementation part), final presentation, presentation slide.

- 黃韋智
  overall ATPG flow, dynamic test compression, podem modification, final report(the implementation part), final presentation, presentation slide.

- Divya Jain
  static test compression(simple), final report.

- 朱宇融
  static test compression(TS), final report.

6. References

- B. Benware, C. Schuermyer, S. Ranganathan, R. Madge, P. Krishna- murthy, ”Impact of multipledetect test patterns on product quality”, IEEE Int’l Test Conference, 2003.

- I.Hamzaoglu, J.Patel, ”Test set compaction algorithms for combinational circuits”, ICCAD 1998.
- Xiang, Dong, et al. ”Compact test generation with an Influence input measure for launch-on-capture transition fault testing”, IEEE Transac- tions on Very Large Scale Integration (VLSI) Systems 22.9 (2014)
- H. Ichihara, A. Ogawa, T. Inoue, A. Tamura, ”Dynamic test compression using statistical codeing”, IEEE Proceedings 10th Asian Test Symposium, 2001.
- Yu-Wei Chen, et al. ”Parallel order atpg for test compaction”, IEEE International Symposium on VLSI Design, Automation and Test, 201