Assessing the Effectiveness of Software-Based Self-Test Programs for Static Cell-Aware Test

Assessing the Effectiveness of Software-Based Self-Test Programs for Static Cell-Aware Test

Riccardo Cantoro*, Michelangelo Grosso†, Iacopo Guglielminetti†, Reza Khoshzaban*, Matteo Sonza Reorda*
* Politecnico di Torino, DAUIN — Turin, Italy † STMicroelectronics — Turin, Italy

***Abstract*—Software-Based Self-Test (SBST) is vastly adopted as a hardware safety mechanism for the in-field test of safety-critical systems in the form of Software Test Libraries (STLs). Typically, an STL's diagnostic coverage is evaluated on the stuck-at fault model. As various defect-oriented fault models exist and are used for manufacturing testing, such as the popular cell-aware test (CAT), there is a need to evaluate the effectiveness of SBST when such models are targeted. This work targets static CAT faults. We evaluated the fault coverage of open-available STLs for a RISC-V SoC. We used results stemming from stuck-at fault simulation and gate-exhaustive simulation to elaborate on the obtained results.**

I. Introduction

Modern electronics in safety-critical systems must meet strict quality requirements. Manufacturing tests are crucial to detect and mitigate hardware defects. Defect-oriented fault models, such as cell-aware testing (CAT)[1]–[3] and small delay defects (SDDs)[4]–[6], are increasingly used in automatic test pattern generation (ATPG) flows to reduce defective parts per million (DPPM) in production[7]. Functional safety standards like ISO26262 mandate a certain quality level, quantified through diagnostic coverage (DC), in the field[8], [9]. However, fault models for fault injection campaigns to quantify coverage metrics are not explicitly referenced in these standards. In-field test solutions still rely mainly on the stuck-at fault model[10], [11]. Functional approaches like software test libraries (STLs) are often adopted in functional safety when hardware solutions are not an option[12]. STLs are based on the well-known software-based self-test (SBST) paradigm [13], which is the main focus of this work. STLs are collections of test programs, often developed by the manufacturing company, that, when run by the CPU inside the Circuit Under Test, allow the detection of a given number of faults by observing the results that are produced. Few works can be found in the literature going in this direction [14], [15].

This work aims to assess the effectiveness of STLs developed for stuck-at faults in detecting static cell-aware faults. As the static CAT fault model is a superset of the stuck-at fault model, we aim to show whether the typical redundancy introduced by test engineers during STL generation is beneficial for detecting cell-aware faults. We present results on a RISC-V core using previously developed and open-source STLs [16]. In our study, we group faults according to the number of input combinations to be applied to the cell to excite and propagate those faults to the cell output(s). As a result, we can identify groups of fault that are potentially harder to test. We analyze the same profile on gate-exhaustive [17] and stuck-at faults to show the peculiar characteristics of CAT. As commercial functional fault simulators that implement CAT in sequential circuits require some pre- and post-processing of the fault lists to assess the CAT fault coverage correctly, in this work, we also present the details of our fault simulation flow. Our experiments show that STLs developed for stuck-at faults provide comparable fault coverage values on static CAT. Moreover, the paper contributes to defining an analysis methodology to understand better the faults that are harder to test using STLs.

II. Background

A. Cell-Aware Test

Traditional fault models like stuck-at, transition delay, bridges, or small-delay defects enumerate faulty locations in the circuit by simply considering the boundary of the standard cells. Such faults are excited by bringing proper logical values (or transitions, in the case of dynamic fault models) toward the faulty locations. Subsequently, the fault differences are propagated through one or more paths by forcing the off-path signals to transparent values (e.g., 1s for AND, 0s for OR gates), until reaching an observable point (e.g., a flip-flop or a primary output). Contrarily, the CAT methodology targets electrical-level defects occurring within the cells, which patterns generated targeting traditional fault models may fail in detecting [1]–[3]. The intra-cell analysis requires working at the electrical level. However, the CAT methodology requires that such extra work is done by characterizing each cell of the technology library independently. Once all cells are analyzed, one can import the resulting cell-aware library in the ATPG or fault simulation flows However, the extra complexity required for CAT ATPG and fault simulation can severely affect the performance. Most commercial Electronic Design Automation (EDA) tools support CAT ATPG and fault simulation for scan design, while the support for functional fault simulation is currently very limited.

Cell-aware characterization begins with *layout extraction* for each cell. A tool analyzes the cell's transistor-level netlist with layout information and extracts a list of possible defects such as parasitic resistors, coupling capacitors, resistive bridges, shorts, or opens. Depending on the EDA vendor, it produces a user-defined fault model (UDFM) or a Cell Test Model (CTM).

In our experiments, we have used the CTM format. The CTM file includes static and dynamic defect matrices for each

Table I: Example of static defect matrix of a full-adder cell

A	B	CI	CO	S	$D1$	$D2$	$D3$	$D4$	$D5$	...
0	0	1	0	1	0	0	1	2	3	...
0	1	0	0	1	3	1	1	2	1	...
1	0	1	1	0	0	2	0	2	0	...
1	1	0	1	0	0	1	0	0	3	...
...	...	...	...	...	...	...	...	...	...	...

cell. Table I shows a snippet of the static defect matrix of a full-adder cell. In the three groups of columns, for each row, the table reports the input values, the expected values on the outputs in the defect-free cell, and a bit-string in decimal format for each defect ($D1$ to $D5$), respectively. The dynamic defect matrices have a similar structure. Still, each line may include dynamic values in the input and output signals (i.e., R and F, for the rising and falling transitions, other than 0 and 1). The bit-string is used to specify whether the defect manifests itself on the corresponding output. In the example, 1 means the defect can be detected on the first output (CO), 2 on the second output (S), and 3 on both. For example, when $D1$ is present in the circuit and the values $[0, 1, 0]$ are applied to the $[A, B, CI]$ inputs of the full-adder, the values on both CO and S are inverted ($[1, 0]$ instead of $[0, 1]$). Using the same input value but injecting $D2$ makes only $C0$ invert its value (i.e., the values on the two outputs are $[1, 1]$ instead of $[0, 1]$). Finally, each defect can be detected by more than one input combination. In the example, $D2$ can be detected by any of the last three test vectors.

B. Software-Based Self-Test

Alternative testing methods have been created to expand testing options for designers and engineers. One such method is SBST, which applies functional stimuli to a microprocessor to detect structural faults. This approach can detect faults without additional power consumption or DfT features.

Previous works addressed SBST generation targeting various fault models such as stuck-ats [18], transition delays [19]–[22], small delays [14], [23] and path delays [24]–[27], while the use of CAT is relatively new [15]. Regarding the methodology for generating test programs, manual approaches, and partly or fully automated techniques have been used based on ATPGs, SAT-solvers, and evolutionary algorithms [13].

III. Fault simulation flow

The SBST fault grading requires simulating the top-level circuit in functional mode while injecting faults and observing their effects on the observable points (e.g., memory elements, the external bus, or specific primary outputs of the target circuit). Fault simulators for functional safety assurance exist, but the native support to CAT is limited. The fault simulator used in our flow supports the extraction of *conditional stuck-at faults* (SAFs) from the static defect matrices of CTM files — more specifically, SAFs on the output pins of a cell with constraints on the cell's inputs. However, such conditional SAFs are specific to a single cell's output and have a fixed polarity (stuck-at-0 or stuck-at-1). As each defect can be excited by some patterns that excite both stuck-at-0 and stuck-at-1 (e.g., $D4$ in the example in Table I), possibly on multiple gate's outputs (e.g., $D1$ in the same example), the fault grading process must consider the defect as tested if any of the multiple conditional SAFs is tested.

The fault simulation approach we propose is based on fault simulating a set of conditional SAFs for each defect identified by the CAT characterization process and then post-processing the resulting fault list to map the stuck-at fault coverage on the CAT defect coverage.

In the technology library used for our experiments (the Silvaco Open-Cell 45nm FreePDK [28]), the cells have only one or two output ports, and the maximum number of faults to inject per defect is four. The cells with only one output port have up to two faults.

The fault simulation produces a fault list that must be post-processed and mapped on the CAT defects. The post-processing phase splits the fault list into sets of conditional SAFs, each one implementing a defect, and checks whether one of the SAFs is marked as detected. In that case, the defect is also marked as detected. If none of the faults is marked as detected, we need to check if any fault is potentially detected; otherwise, the defect is marked as not detected. In more complex fault grading processes (e.g., for functional safety assurance), where additional tags can be used to mark faults, one can adapt the post-processing flow by adding additional conditions.

IV. Experimental results

We have assessed the effectiveness of SBST programs concerning static CAT faults on the RI5CY processor, a 4-stage 32-bit RISC-V in-order RISC-V processor core embedded in the PULPino SoC [29]. We synthesized the design using the Silvaco Open-Cell 45nm FreePDK [28]. We used Synopsys Design Compiler for the logic synthesis, Synopsys CMGen for the CAT characterization process, Siemens QuestaSim for logic simulation, and Synopsys Z01X for fault simulation. All the experiments have been run on an Intel Xeon CPU E5-2680 v3 machine. As we have experienced some issues simulating faults affecting sequential cells due to some limitations of the available tools, we have restricted our analysis to the defects affecting the combinational cells. For the sake of comparison, the results of the stuck-at fault (SAF) and gate-exhaustive (GE) models do not include faults in the sequential cells.

Table II reports the number of SAFs (column 2), conditional SAFs mapping the CAT defects before the post-processing (*Cond*, column 3), CAT defects after the post-processing (column 4), and gate-exhaustive (*GE*, column 5) for the main modules of the core. GE faults have been implemented using conditional SAFs, similar to what was done for CAT defects. The number of CAT defects is three to four times that of SAFs. The significant differences between pre- and post-processing are visible on the ALU, the multiplier, and the load/store unit. The post-processing step has collapsed the total number of faults by almost 15%.

Table II: Faults in the RI5CY core (combinational cells only)

Module	#SAF	#Cond	#CAT	#GE
CS regs	6,158	17,537	17,537	8,010
Debug	2,596	7,093	7,093	3,268
EX/ALU	25,754	88,611	81,075	34,438
EX/MUL	30,930	204,525	144,682	48,116
ID/Control	1,322	3,593	3,593	1,454
ID/Decoder	3,422	9,219	9,219	3,870
ID/HW loop regs	3,332	12,714	12,714	4,100
ID/INT control	92	379	379	110
ID/regs	33,324	97,880	97,880	49,520
IF/compr, dec,	1,476	3,959	3,959	1,694
IF/HW loop control	1,628	4,449	4,446	2,636
IF/Prefetch buffer	9,648	27,134	27,133	13,270
Load Store	4,054	16,069	13,488	6,354
CPU (TOP LEVEL)	141,990	554,118	480,362	206,718

Our analysis uses a set of open-available stuck-at STLs [16]. The cell-aware fault simulation lasts much longer than the stuck-at fault simulation, ranging from 75x runtime for STL6 (5 hrs against 4 mins) to 422x for STL5 (204 hrs against 29 mins).

Our experiments show that STLs can achieve significant fault coverage values on static CAT, even when the target fault model is SAF (see Table III). Remarkably, all STLs can detect more than 80% of static CAT faults, with a maximum of almost 87% reached by STL5. Moreover, as shown by the results, the fault coverage would be overestimated without the post-processing step. It is interesting to note that STLs are not as effective as on CAT faults when evaluating them on GE faults, with a fault coverage between 43% and 47%. This was expected, as testing each GE fault requires a specific input to be applied to a logic gate, while each CAT defect often has multiple detection conditions.

To deeply understand the reasons behind the high fault coverage, we have performed an accurate analysis on the CAT fault lists, aimed at verifying if there are correlations between the number of lines in a cell's truth table that can be used to detect the fault and the actual fault coverage obtained by the STLs. We also performed the same analysis on SAF and GE faults. We grouped all faults in the fault list detectable by the same percentage of lines in their relative cell's truth table (e.g., all faults detected by one out of four lines). We denoted this percent figure as *table test*% ($TT\%$). Then, we computed the size of each group and the fault coverage values. The results for CAT have been reported in Table IV, ordered by $TT\%$.

The analysis shows that groups with low $TT\%$ for CAT are smaller than the SAF ones. In general, CAT groups have more variability in size than SAF groups.

The analysis shows that defects with low $TT\%$ are harder to test. This was expected, but we can now quantify the impact of these faults on the final coverage. As a significant result, we observed how the impact on the final fault coverage of the critical groups is lower on CAT faults than on SAFs.

V. Conclusions

This paper presented an analysis to assess the effectiveness of some existing STLs in detecting static defects modeled according to the popular cell-aware test (CAT) approach. We first described the challenges to be faced for implementing a correct fault grading process, such as (i) simulating static CAT defects by resorting to a functional fault simulator able to extract conditional stuck-at faults (SAFs) from a CAT defect matrix and (ii) post-processing the fault simulation results to map faults on the corresponding defects. Experimental results on a RISC-V core have shown that a fault grading process can be successfully implemented at the cost of a considerable penalty in performance. Interestingly, the evaluated STLs were able to cover up to 87% of CAT faults on the CPU module in our experiments. We also proposed a metric able to predict which cells and defects are harder to test, thus paving the way to devising solutions able to guide the improvement of existing STLs in order to enhance the achieved CAT FC. In the future, we plan to extend our work to also consider dynamic CAT faults.

Acknowledgment

This publication is part of the project PNRR-NGEU which has received funding from the MUR – DR 117/2023

References

[1] F. Hapke *et al.*, "Defect-oriented cell-aware atpg and fault simulation for industrial cell libraries and designs," in *IEEE Int'l Test Conf.*, 2009.

[2] F. Hapke *et al.*, "Defect-oriented cell-internal testing," in *IEEE Int'l Test Conf.*, 2010.

[3] F. Hapke *et al.*, "Cell-aware analysis for small-delay effects and production test results from different fault models," in *IEEE Int'l Test Conf.*, 2011.

[4] M. Yilmaz *et al.*, "Test-pattern grading and pattern selection for small-delay defects," in *IEEE VLSI Test Symp.*, 2008.

[5] S. K. Goel *et al.*, "Effective and efficient test pattern generation for small delay defect," in *IEEE VLSI Test Symp.*, 2009.

[6] S. K. Goel *et al.*, *Testing for Small-Delay Defects in Nanoscale CMOS Integrated Circuits*, 1st. USA: CRC Press, Inc., 2013.

[7] F. Hapke *et al.*, "Introduction to the defect-oriented cell-aware test methodology for significant reduction of dppm rates," in *IEEE European Test Symp. (ETS)*, 2012.

[8] ISO/TC 22/SC 32, "Road vehicles – functional safety – Part 1-12," en, International Organization for Standardization, Geneva, Switzerland, Standard ISO 26262-1:2018, 2018.

[9] A. Nardi *et al.*, "Functional safety methodologies for automotive applications," in *IEEE/ACM Int'l Conf. on Computer-Aided Design*, 2017.

[10] A. Nardi *et al.*, "Design-for-safety for automotive ic design: Challenges and opportunities," in *IEEE Custom Integrated Circuits Conference*, 2019.

[11] F. A. da Silva *et al.*, "An automated formal-based approach for reducing undetected faults in iso 26262 hardware compliant designs," in *IEEE Int'l Test Conf.*, 2021.

[12] F. Pratas *et al.*, "Measuring the effectiveness of iso26262 compliant self test library," in *Int'l Symp. on Quality Electronic Design (ISQED)*, 2018.

[13] M. Psarakis *et al.*, "Microprocessor software-based self-testing," *IEEE Design & Test of Computers*, vol. 27, no. 3, 2010.

[14] A. Riefert *et al.*, "An effective approach to automatic functional processor test generation for small-delay faults," in *Automation & Test in Europe Conference & Exhibition (DATE)*, 2014.

[15] P. Bernardi *et al.*, "Recent trends and perspectives on defect-oriented testing," in *IEEE Int'l Symp. on On-Line Testing and Robust System Design*, 2022.

[16] CAD Group, Politecnico di Torino, *Stuck-At STLs for pulpino-ri5cy*, https://github.com/cad-polito-it/pulpino_ri5cy_stls, 2023.

[17] K. Y. Cho *et al.*, "Gate exhaustive testing," in *IEEE Int'l Conf. on Test*, 2005.

[18] D. Gizopoulos *et al.*, "Systematic software-based self-test for pipelined processors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 11, 2008.

Table III: Fault simulation results

Module	STL3 FC%				STL4 FC%				STL5 FC%				STL6 FC%				STL7 FC%			
	SAF	Cond	CAT	GE	SAF	Cond	CAT	GE	SAF	Cond	CAT	GE	SAF	Cond	CAT	GE	SAF	Cond	CAT	GE
CS regs	66.3	71.5	71.5	36.5	46.2	52.4	52.4	20.6	62.6	69.3	69.3	32.1	72.7	74.0	74.0	34.6	54.6	63.9	63.9	27.3
Debug	16.1	25.8	25.8	8.6	16.5	25.8	25.8	8.9	16.7	26.0	26.0	9.1	16.5	25.8	25.8	8.9	18.4	25.8	25.8	10.9
EX/ALU	78.6	79.2	79.9	43.0	82.3	86.0	84.7	47.1	91.4	92.7	92.2	53.8	84.0	84.3	84.5	47.0	87.5	90.2	89.0	53.7
EX/MUL	93.8	88.3	90.1	51.1	95.6	96.9	95.1	67.8	96.2	96.3	94.9	68.4	96.3	95.2	94.4	59.8	98.5	98.7	97.0	66.8
ID/Control	53.8	57.9	57.9	28.0	44.3	47.5	47.5	22.4	54.2	58.5	58.5	28.6	54.0	58.4	58.4	28.2	51.8	56.1	56.1	26.7
ID/Decoder	81.2	81.4	81.4	48.0	73.0	73.7	73.7	42.0	83.5	83.7	83.7	49.8	82.5	82.9	82.9	49.4	81.3	81.3	81.3	48.3
ID/HW loop regs	63.6	58.5	58.5	35.5	79.3	83.2	83.2	52.8	71.3	70.3	70.3	46.6	77.7	65.2	65.2	44.0	46.1	46.2	46.2	28.7
ID/INT control	20.7	25.6	25.6	9.1	10.9	15.8	15.8	4.5	20.7	25.6	25.6	9.1	20.7	25.6	25.6	9.1	10.9	15.8	15.8	4.5
ID/regs	91.3	97.0	97.0	47.1	84.8	91.2	91.2	47.1	84.2	90.4	90.4	46.8	82.7	95.0	95.0	47.3	82.2	87.0	87.0	46.4
IF/compr. dec.	40.8	46.7	46.7	21.1	49.6	54.5	54.5	27.3	75.8	76.8	76.8	45.7	68.7	71.6	71.6	40.1	41.1	46.8	46.8	21.3
IF/HW loop contr.	51.2	55.4	55.4	16.4	53.4	58.3	58.3	17.4	57.2	61.6	61.6	19.6	49.2	52.5	52.5	14.6	50.6	54.0	54.0	14.7
IF/Prefetch buf.	67.6	70.9	70.9	34.7	73.4	76.0	76.0	39.6	74.3	76.3	76.3	39.9	72.3	74.8	74.8	38.2	73.1	75.5	75.5	36.0
Load Store	77.4	85.7	82.1	41.7	84.6	92.0	89.6	47.1	85.0	92.6	90.3	47.6	77.8	84.7	80.9	41.5	70.6	82.4	78.2	40.1
CPU (TOP LEVEL)	80.7	82.9	83.1	42.6	80.1	86.6	84.8	47.4	82.9	88.2	86.7	49.3	81.4	86.7	85.7	45.8	80.2	86.7	84.5	47.3

Table IV: Fault simulation results on groups of CAT faults. Each group includes the faults with the same percentage of lines in a faulty cell's truth table able to test them at the cell level ($TT\%$). For each group of faults, we report the fault coverage within the group (*Fault coverage / grouped faults*) — which gives an idea about how easy it is to cover faults in that group — and the fault coverage over all faults (*Fault coverage / all faults)* — which represents the delta value added to the fault coverage when including faults in that group. Cumulative sum (*cumsum*) values are reported for grouped faults' size and fault coverage over all faults. The reader can refer to the *Size% cumsum* value to know the percentage of faults considered up to that line (until 100% in the last line) and read the corresponding cumulative fault coverage reported on the last group of columns (*Fault coverage % cumsum*). The last lines on those columns report the fault coverage reached by the STL on the full fault lists.

Grouped faults info			Fault coverage / grouped faults (%)					Fault coverage / all faults (%)					Fault coverage / all faults (% cumsum)				
TT%	Size%	Size% cumsum	STL3	STL4	STL5	STL6	STL7	STL3	STL4	STL5	STL6	STL7	STL3	STL4	STL5	STL6	STL7
Cell-Aware Test																	
6.25	3.81	3.81	81.42	82.77	89.19	83.68	82.77	3.10	3.15	3.40	3.19	3.15	3.10	3.15	3.40	3.19	3.15
9.38	1.20	5.01	58.73	59.48	67.16	60.61	63.87	0.70	0.71	0.80	0.73	0.77	3.81	3.87	4.20	3.91	3.92
12.50	10.14	15.14	68.33	77.64	79.24	73.48	77.71	6.93	7.87	8.03	7.45	7.88	10.73	11.74	12.24	11.36	11.80
14.06	2.25	17.40	83.00	90.08	89.66	84.52	84.78	1.87	2.03	2.02	1.91	1.91	12.60	13.77	14.26	13.27	13.71
18.75	22.08	39.48	81.25	78.68	80.19	82.89	75.90	17.94	17.37	17.71	18.30	16.76	30.55	31.14	31.96	31.57	30.47
25.00	17.20	56.68	79.09	83.75	85.62	84.36	85.09	13.60	14.40	14.73	14.51	14.63	44.15	45.54	46.69	46.08	45.10
28.13	0.70	57.38	79.90	84.13	86.07	86.91	77.75	0.56	0.59	0.60	0.61	0.54	44.70	46.13	47.29	46.69	45.64
32.81	0.06	57.44	70.59	89.62	98.27	61.94	58.48	0.04	0.05	0.06	0.04	0.04	44.75	46.18	47.35	46.72	45.68
37.50	6.43	63.87	86.03	89.24	93.31	90.23	91.91	5.53	5.74	6.00	5.80	5.91	50.28	51.93	53.35	52.53	51.59
40.63	0.17	64.04	73.76	75.74	79.21	72.77	79.70	0.12	0.13	0.13	0.12	0.13	50.41	52.05	53.48	52.65	51.72
42.19	1.69	65.73	97.66	98.69	98.18	97.92	97.92	1.65	1.67	1.66	1.66	1.66	52.06	53.72	55.14	54.31	53.38
43.75	0.80	66.53	86.21	85.20	88.09	84.49	83.55	0.69	0.68	0.70	0.67	0.67	52.75	54.40	55.85	54.98	54.05
50.00	15.00	81.53	86.88	87.20	88.32	87.61	87.53	13.03	13.08	13.25	13.14	13.13	65.78	67.48	69.10	68.12	67.18
56.25	11.64	93.17	97.56	96.45	98.16	97.97	97.29	11.36	11.23	11.43	11.41	11.33	77.14	78.71	80.53	79.53	78.51
57.81	0.10	93.27	75.71	91.90	98.38	70.85	61.13	0.08	0.09	0.10	0.07	0.06	77.21	78.81	80.63	79.60	78.57
62.50	1.23	94.50	83.60	90.86	94.52	89.90	87.63	1.03	1.12	1.16	1.11	1.08	78.24	79.92	81.79	80.71	79.65
68.75	0.10	94.60	93.83	91.49	90.21	95.53	82.13	0.09	0.09	0.09	0.09	0.08	78.33	80.01	81.88	80.80	79.73
71.88	0.47	95.07	77.91	84.29	84.43	80.37	81.35	0.36	0.39	0.39	0.37	0.38	78.70	80.41	82.27	81.18	80.11
75.00	2.00	97.07	89.95	87.60	90.78	91.50	90.33	1.80	1.75	1.82	1.83	1.81	80.50	82.16	84.09	83.01	81.91
81.25	0.99	98.06	84.19	86.23	89.20	88.64	90.72	0.83	0.85	0.88	0.88	0.90	81.33	83.01	84.97	83.88	82.81
87.50	0.78	98.84	81.69	85.03	86.12	86.34	82.99	0.64	0.66	0.67	0.67	0.65	81.97	83.68	85.64	84.56	83.46
93.75	1.16	100.00	94.61	93.47	94.41	96.19	86.34	1.10	1.09	1.10	1.12	1.00	83.07	84.76	86.74	85.68	84.46

[19] R. Cantoro *et al.*, "Effective techniques for automatically improving the transition delay fault coverage of self-test libraries," in *IEEE European Test Symp.*, 2022.

[20] R. Cantoro *et al.*, "Self-test libraries analysis for pipelined processors transition fault coverage improvement," in *IEEE Int'l Symp. on On-Line Testing and Robust System Design*, 2021.

[21] M. Grosso *et al.*, "Software-based self-test for transition faults: A case study," in *IFIP/IEEE Int'l Conf. on Very Large Scale Integration (VLSI-SoC)*, 2019.

[22] K.-H. Chen *et al.*, "Automatic test program generation for transition delay faults in pipelined processors," in *IEEE Int'l Test Conf. in Asia*, 2021.

[23] M. Grosso *et al.*, "Software-based self-test for delay faults," in *VLSI-SoC: New Technology Enabler*, C. Metzler *et al.*, Eds., Cham: Springer International Publishing, 2020.

[24] Singh *et al.*, "Software-based delay fault testing of processor cores," in *Test Symp.*, 2003.

[25] K. Christou *et al.*, "A novel sbst generation technique for path-delay faults in microprocessors exploiting gate- and rt-level descriptions," in *IEEE VLSI Test Symp.*, 2008.

[26] C.-P. Wen *et al.*, "On a software-based self-test methodology and its application," in *IEEE VLSI Test Symp.*, 2005.

[27] L. Anghel *et al.*, "Self-test library generation for in-field test of path delay faults," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 2023.

[28] Silvaco, *Open-Cell 45nm FreePDK*, https://si2.org/open-cell-library/.

[29] ETH Zurich and Università di Bologna, *PULPino microcontroller system*, https://github.com/pulp-platform/pulpino, 2022.