Maximizing the Switching Activity of Different Modules Within a Processor Core via Evolutionary Techniques

Maximizing the Switching Activity of Different Modules Within a Processor Core via Evolutionary Techniques

Nikolaos I. Deligiannis, Riccardo Cantoro, Matteo Sonza Reorda
Politecnico di Torino, Dip. Automatica e Informatica. Torino, Italy
{nikolaos.deligiannis|riccardo.cantoro|matteo.sonzareorda}@polito.it

***Abstract*—One key aspect to be considered during device testing is the minimization of the switching activity of the circuit under test (CUT), thus avoiding possible problems stemming from overheating it. But there are also scenarios, where the maximization of certain circuits' modules switching activity could be proven useful (e.g., during Burn-In) in order to exercise the circuit under extreme operating conditions in terms of temperature (and temperature gradients). Resorting to a functional approach based on Software-based Self-test guarantees that the high induced activity cannot damage the CUT nor produce any yield loss. However, the generation of effective suitable test programs remains a challenging task. In this paper, we consider a scenario where the modules to be stressed are sub-modules of a fully pipelined processor. We present a technique, based on an evolutionary approach, able to automatically generate stress test programs, i.e., sequences of instructions achieving a high toggling activity in the target module. With respect to previous approaches, the generated sequences are short and repeatable, thus guaranteeing their easy usability to stress a module (and increase its temperature). The processor we used for our experiments is the Open RISC 1200. Results demonstrate that the proposed method is effective in achieving a high value of sustained toggling activity with short (3 instructions) and repeatable sequences.**

I. Introduction

Efficient test solutions are crucial to guarantee the target reliability figures in an electronic system. Test of electronic devices is performed in several different steps. The most difficult task for a test engineer lies in selecting the right mix, allowing to achieve the reliability targets with acceptable costs. The whole set of test steps usually adopted by semiconductor companies at the end of the manufacturing process may include Burn-In (BI), whose goal is to artificially age the Circuit Under Test (CUT) so that any weak point evolves into an observable fault and can be detected. In this way, the phenomenon known as *Infant Mortality* is greatly reduced and does not impact the failure rate of the delivered devices, which can be lowered and remains stable during the operational life.

In the past, BI was typically based on stressing the CUT resorting to high temperature and voltage. More recently, the high duration and cost of such procedures are making them increasingly unaffordable. Tuning the key parameters of BI (duration, temperature, voltage) is becoming more difficult since it requires a long characterization phase for each new technology; at the same time, process variations introduce a significant amount of uncertainty. In this scenario, BI is evolving to new forms, where the stress is created with less dangerous and more controllable actions, e.g., resorting to *internal* stress. In this case, the target stress is induced in the CUT by forcing it to repeatedly execute operations that can intensively activate the internal structures, e.g., by maximizing the toggling activity [1]. In this way we can not only force transistors inside the CUT to toggle at a high rate, but also increase the CUT temperature. Both effects are known to contribute to quick aging, thus achieving the BI targets. In some cases, a high toggling activity can be achieved resorting to the existing scan infrastructures. ATPG tools can be requested to generate test vectors able to maximize the toggling activity. However, since scan forces the CUT to work in a configuration different than the operational one, care must be taken not to *overstress* the circuit, possibly causing yield loss. To avoid this risk, some researchers in the recent past focused on adopting purely functional solutions [2]. In this case, the circuit works in normal mode, and the CPU is forced to execute some specially crafted programs to maximize the toggling activity.

Maximizing the toggling activity, either in the whole CUT or in some parts of it, may turn to be effective even in other test steps. For example, it has been speculated that testing for delay faults while the circuit temperature is at the top of the allowed range may permit detecting a higher percentage of defects [3] [4].

More recently, the adoption of System Level Test (SLT), to detect defects that could escape all the traditional test steps, leads to the search for functional stimuli able to produce particularly stressful conditions. Once again, this can be achieved by maximizing the internal activity and/or creating temperature gradients between different modules within the CUT [5].

Reliability of integrated circuits is one of the key attributes of testing. A well-known testing practice to enhance the device reliability and screen out early failures (i.e., Infant Mortality) in device components is BI. During BI, the device is exercised under elevated temperature and power conditions. Work has been done in the past to maximize the heat [6] and power [7] dissipation of the devices during BI. In [8] a method based on formal techniques is proposed, concerning the generation of stress vectors to be used as stimuli during BI. As regards processor testing, an evolutionary-based technique has been suggested in [2] to identify and extract sequences of

instructions in already existing test programs to maximize the switching activity of certain modules.

To summarize, different steps are currently adopted for the test of current devices, where the task of generating test programs able to maximize the toggling activity either in the whole CUT or in a specific target module is crucial.

This paper formalizes this problem and describes a method to generate such maximum activity test programs for different modules within a CPU. With respect to previous solutions, the generated test programs not only maximize the switching activity but can also be repeated, thus guaranteeing that a high value is achieved over a period of any duration. In this way, we can really use the generated test programs to control the CUT temperature. Results gathered on an OpenRISC 1200 (OR1200) pipelined processor show that the method can produce high-quality test programs with an acceptable computational effort. The method is relatively easy to be adopted since it does not require in-depth knowledge of the processor's architecture, but a rather good knowledge of the instructions that compose the processor's Instruction Set Architecture (ISA). Also, the method generates stress programs from the ground up and does not require any dependency on pre-existing programs. Furthermore, the generated test programs are relatively short in length, typically composed of 3 instructions.

The rest of the paper is organized as follows: Section II concerns the formalization of the problem and provides the concept idea. In Section III we present and elaborate on the proposed method. In Section IV we elaborate on the implementation details of the method. In Section V we provide the experimental setup along with the respective results. Finally, in Section VI we draw some conclusions and provide insight on our future work.

II. Problem Definition

In this paper we propose a method, based on evolutionary techniques, for the automatic generation of assembly programs i.e., sequences of instructions able to maximize the switching activity of a certain module (or sub-module). Since we want to be able to maximize the toggling activity over a large number of clock cycles (i.e., instructions) in a sustained manner, we also enforce a further constraint on the generated sequence, which must be **repeatable**. This means that, assuming that the target module starts from an initial state $\sigma_{initial}$, and that after the execution of the generated sequence moves to a final state σ_{final}, it must hold that $\sigma_{final} \equiv \sigma_{initial}$. By guaranteeing that a sequence is repeatable, we can maintain a sustained high nodal activity within the target module for an arbitrarily long period of time.

More specifically, given a sequence $\widetilde{s}_n$ of n instructions and a processor module composed of m nets, we aim at maximizing the value of the induced switching activity on the module's nets by applying $\widetilde{s}_n$. Since the problem can be seen as an optimization problem, we define our objective function as:

$$SW = \frac{1}{n \times m} \sum_{i=0}^{m} t_{i,\widetilde{s_n}} \tag{1}$$

$t_{i,\widetilde{s_n}}$: the number of logical switches performed by net i while $\widetilde{s}_n$ was executed

Equation (1) represents the normalized switching activity for a sequence of instructions $\widetilde{s}_n$. The total nodal activity for the module m is given by the sum and is normalized to the $[0.0, 1.0]$ range by dividing it with the theoretical maximum given by the fraction $1/(n \times m)$; meaning for every net of the considered processor module to toggle when each instruction of the sequence $\widetilde{s}_n$ is executed. This value, which is considered to be the maximum, in practice may be affected negatively by the presence of uncontrollable lines inside the module [9]. Thus, it can be interpreted as a theoretical maximum. Additionally, we further require from the aforementioned sequence to be repeatable.

The problem of the maximization of the switching activity for a specific module of the processor can be now defined as:

$$\max_{\widetilde{s}_n} \{SW\} \tag{2}$$

While the goal is the same for every processor module, namely, increasing the module's switching activity over a period of time by forcing the execution of a suitable chunk of instructions, the complexity of the approach differs according to the module. Let us assume for example that we wish to stress the circuitry of the adder, which is located in the processor's arithmetic and logic unit. In that case, the algorithm would have to identify pairs of operands that would maximize the toggling activity of the adder, since the assembly instruction that must be used on this scenario (i.e., add) is known.

Hence, the couples of operands for the add instruction we are looking for are supposed to maximize the toggling activity induced when executing the i-th instruction with respect to the state created by the $i-1$ instruction. If for example we were focusing on the generation of a sequence composed of $n = 2$ instructions, this would mean that the algorithm would have to identify pairs of operands for maximizing the switching activity during the transition from the first to the second instruction and from the second instruction to the first. Also, since the sequence is also a repeatable one, it should hold that the final state, in which the circuit is driven after the execution of the second instruction, should be equal to the initial state. But if on the other hand we were interested in the generation of sequences of composed of $n > 2$ instructions, then the complexity of the search would be higher since we would impose more maximizations during transitions between instructions to be considered by the method.

On the other hand, assuming that we now wish to stress the processor's instruction decode unit, then the algorithm should further consider the whole set of instructions and not only the operands to be used in order to stress the module. In the

case of the latter, the complexity of the task increases, since the search space of the algorithm is now bigger than in the former case.

Let us assume that a repeatable stress sequence was generated from the algorithm. Since that sequence is independent, meaning that it does not require any preliminary instructions to setup the execution environment, one example of its application to the processor (e.g., to increase its temperature) can be via loop unrolling. We repeat the sequence a discrete and finite amount of times and at the end of the unrolled segment we place an unconditional $jump$ to transfer the code execution back to the start. In this way, the generated stress sequence can be applied an arbitrarily large number of times until an interrupt is issued.

III. Evolutionary Approach Concept

As mentioned in Section II we aim at identifying a short sequence of instructions able to maximize the switching activity of a given processor's module. For the purposes of this task, we developed an algorithm based on the evolutionary paradigm. Given a well defined problem, the evolutionary algorithms generate and propose solutions in a manner inspired by nature. The approach can be seen as an optimization of a mathematical function. The algorithm initially generates solutions to the problem i.e., *individuals* in a random manner. The generated individuals compose a *generation*. Every individual of the population is assigned a *fitness* value. The fitness is a function which takes as input the individual and returns a value according to how "good" this individual is with respect to the problem consideration. After the assignment of the fitness values a ranking of the population takes place in order to distinguish between the "good" and the "bad" solutions. The individuals with the better fitness values are undergoing a selection process to become parent individuals and to produce offsprings, which will be part of the next generation. The generation of new individuals is primarily the result of the application of the *genetic operators* on the parent individuals. One of the most common genetic operator is the cross-over, during which, the new individuals are generated from the splicing of the parents characteristics. Finally, the *mutation* procedure takes place, which is a probabilistic alternation on the characteristics of the individuals.

The proposed algorithm takes as input a set of constraints and is able to produce individuals, i.e., **assembly programs** for the target processor. Given the gate-level description of the processor and a certain module that we wish to stress, we obtain individuals and compute the stress they cause on a certain processor module via logic simulation. Specifically, we can aggregate the overall nodal activity, meaning the number of HL (High to Low) and LH (Low to High) transitions that were performed by every net of the processor's module during the execution of the stress program.

The stress program generation procedure is summarized in Figure 1. The routine takes as input parameters a triplet, a set of population settings (S), a set of population constraints (C) and an integer that is used as a stop criterion by the algorithm (Max).

```
input  : A triplet (S,C,Max) where
         S is the set of population settings
         C is the set of population constraints
         Max is the maximum allowed steady generations
output : A collection of assembly programs L
1  best_old := null
2  best_new := null
3  steady_cc := 0
4  do
5      L     ← GenerateIndividuals(S,C)
6      L_FIT ← FitnessEvaluation(L)
7      best_new = FindMaxFitnessValue (L,L_FIT)
8      if best_old == best_new then
           // best fitness didn't change
9          steady_cc = steady_cc + 1
10     end
11     else
           // new best fitness
12         steady_cc = 0
13     end
14     best_old = best_new
15 while (steady_cc ≠ Max)
16 return best_old
```

Fig. 1: Stress Program Generation Routine

A. Fitness Function and Metrics

Until the halt criterion is met, individuals are generated for the maximization of the switching activity of the defined processor module. A logic simulation for every individual is performed in order to evaluate the stress that the program induces on the processor's module in order to assign a fitness value to it.

Given that the module of the processor consists of m nets and the assembly program consists of n instructions, we define the fitness function for the individual X as:

$$\textit{fitness}(X) := \frac{\sum_{i=1}^{m}[HL(i) + LH(i)]}{n \times m} \tag{3}$$

and it holds:

$$\textit{fitness}(X)\ \varepsilon\ [\mathbf{0.0}, \mathbf{1.0}]$$

$HL(i)/LH(i)$ is a function that takes as input the index (i) of the processor's module net and returns the total amount of HL/LH transitions that line performed during the execution of the test program. The nominator of Equation (3) represents the switching activity induced by the program, while the denominator represents the maximum, meaning for every net of the processor's module to toggle when each instruction is executed.

For the best individuals generated from the procedure for every processor module considered, we introduce the following metrics:

- $HL \wedge LH$: The percentage of nets that performed both transitions at least once.

- HL: The percentage of nets that performed only the HL transition.
- LH: The percentage of nets that performed only the LH transition
- $\neg HL \wedge \neg LH$: The percentage of nets that did not perform any transition.

IV. Implementation Details

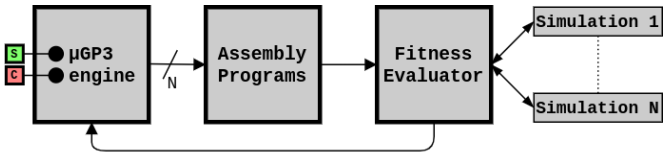

Fig. 2: Proposed Approach Implemented with μGP3

In order to implement the proposed method we developed a framework that utilizes the μGP [10] (version 3) as an underlying engine. μGP is an evolutionary optimizer that was initially developed to produce assembly programs maximizing a given fitness function for a variety of processors. It takes as input a user-defined set of rules and parameters (e.g., sets S, C of fig. 1) and initially provides random solutions that get refined during the evolution process. Furthermore, the tool depends heavily on an external module that must be also given as a parameter to the engine and performs the fitness evaluation of the generated individuals. The external fitness evaluator is a user-provided module responsible of describing the fitness function and providing a fitness value for every individual to the evolutionary core.

Figure 2 illustrates the flow of our experiments. The first step is to define a target, namely, the processor module we wish to stress in order to generate the two sets S and C, which are given as input to the tool. Next, the evolution process begins. On every iteration a population of individuals is generated based on an amount of genetic operations and mutations taking place. Then, a call on the external fitness evaluation takes place in order to assign a fitness value to the individuals and proceed to the next evolutionary step. Besides the two configuration sets (S and C), we further specify a number (Max) to the algorithm, which is used as a termination criterion. Specifically, on every new generation we observe if the best fitness value has changed. If the best fitness value remains unchanged Max times, then the algorithm halts and returns the best individual.

A. Population Settings (Set S)

The first of the two sets given to the evolutionary tool as input corresponds to the population settings. It is a collection of parameters that are linked directly to the genetic algorithm such as the population size, the number of genetic operators to be applied on each iteration, and so on. A detailed description of the basic parameters along with their respective values on our experiments are reported in Table I below.

The number of individuals to be generated on the initial population during the first iteration is defined by the parameter ν. On every evolutionary step forward λ individuals are being generated by the application of genetic operators and mutation operations. The impact the latter has on the individuals is defined by the parameter σ. After the ranking of the individuals an elimination phase on the population follows in order to comply with the size limit defined by the parameter μ. Also, if a certain individual is within the μ limit, and is not substituted by any of his successors, he gets forcibly eliminated after $maxAge$ generations. The parameter α defines the rate of change of the internal evolutionary core parameters. It is set to a high value in order to prohibit random changes on the self-adapting parameters of the tool. Lastly, we protect from the forced elimination after the $maxAge$ generations $elite$ individuals. These individuals are kept alive until new, improved individuals replace them.

TABLE I: Population Settings

Parameter	Context	Value
ν	Initial size of the population	320
μ	Maximum size of the population	200
λ	Number of genetic operators per every step	120
σ	Strength of mutation operators	0.9
α	Inertia of the self adapting parameters	0.9
$maxAge$	After this limit the individual is forcibly killed	15
$elite$	Number of best individuals that do not get killed	2

B. Constraints Settings (Set C)

The set of constraints given as the second parameter to the tool regards the set of rules and formats the tool has to consider in order to generate valid individuals. In other words, to ensure that the tool will generate syntactically correct assembly programs. Although the population settings remains unchanged in our experiments, the set of constraints has to be tailored according to the processor module that we aim to stress. In this paper, we are focused on three modules:

- The $adder$
- The $multiplier$
- The $instruction\ decode$ unit

This means that three separate sets must be generated. Besides ensuring syntactical correctness on the generated solutions, the constraints are also used to guarantee that the generated sequences of instructions will be repeatable. Let us assume that we aim to stress the $adder$ module. Assuming that there is a rule within the constraints that mandates the generation of a sequence of n add instructions $\widetilde{s} = < add_0, ..., add_{n-1} >$ we further specify that the generated individual must have the following structural property: $\widetilde{s}, \widetilde{s}, ..., \widetilde{s}$. Meaning, that the individual will be composed by the repetition of the sequence $\widetilde{s}$ a discrete amount of times.

C. External Fitness Evaluator

The external fitness evaluator is responsible for receiving a list of individuals and providing back to the evolutionary core their respective fitness values. In our case, the evaluator has to calculate the fitness value according to the Equation (3). The

TABLE II: Experimental Results

Performance of Best Individuals						
Processor Module	**Fitness (%)**	**Clock Cycles**	**HL ∧ LH**	**HL**	**LH**	**¬HL ∧ ¬LH**
Adder	61.34%	113	72.02%	0.00%	0.00%	27.98%
Multiplier	54.77%	137	74.92%	0.00%	0.09%	25.08%
Decoding Unit	62.57%	93	75.11%	0.00%	0.00%	24.89%

Performance of Test Program Segments					
Processor Module	**Fitness (%)**	**HL ∧ LH**	**HL**	**LH**	**¬HL ∧ ¬LH**
Adder	24.00%	31.51%	3.36%	0.00%	65.13%
Multiplier	6.34%	21.78%	6.59%	6.97%	64.66%
Decoding Unit	44.43%	15.23%	0.57%	2.50%	81.7%

nominator value is calculated by launching a logic simulation for every individual in order to determine the exact amount of logical switches.

V. Experimental Setup and Results

All the experiments were performed on a machine using 2 Intel Xeon CPUs running at 2.40GHz. Since the evaluation of every individual can be performed in a parallel manner, we parallelized each experiment by assigning 20 cores to our external fitness evaluator. For the purposes of logic simulation we used QuestaSIM by Mentor Graphics.

The processor used in our experiments is the OR1200. The OR1200 is a 32-bit scalar RISC with Harvard micro-architecture and 5 stage integer pipeline. The OR1200 core is mainly intended for embedded, portable and networking applications. The RT-level description of the core from [11] was synthesized using the Silvaco 45nm Open Cell Library [12].

As mentioned before, in this paper we focus on the $adder$ the $multiplier$ and the $decoding\ unit$ of the processor. The modules were selected in that order, in order to show that the method can be applied to simple modules (e.g., $adder$), but also in more complex modules such as the $decoding\ unit$ of a processor. As a compromise between complexity, in terms of CPU time, and performance, in terms of induced stress in the aforementioned modules, we selected the length of the repeatable stress sequences to be 3.

Approximately 500 lines of code were written in $bash$, tcl and $python$, which account for the external fitness evaluator, the logic simulations, the linking with the evolutionary core and the data extraction.

The progress of the evolution for each experiment is illustrated in Figure 3. We can see that the algorithm converged for the $adder$ and the $multiplier$ after approximately 700 generations while for the $decoder$ after approximately 1,200 generations. The plotted lines show the behavior of the fitness value of the best individuals on each generation. The experiments for the $adder$ and the $multiplier$ required approximately 19 hours to converge, while for the case of the $decoding\ unit$ the algorithm converged in 24 hours.

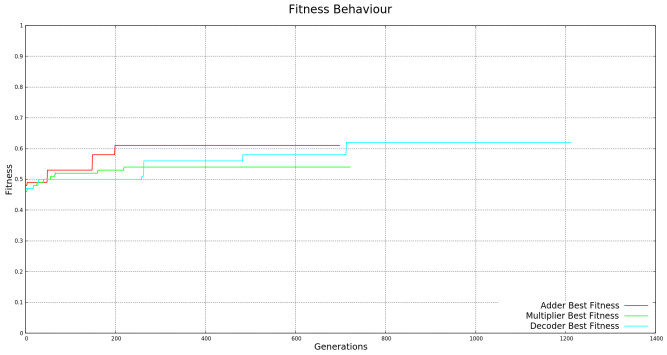

Fig. 3: Fitness Behaviour per Generation

Table II reports the results of our experiments. The upper half of the table lists the performance of the best individuals generated from the evolutionary tool for every processor module. The total stress induced on the modules of the $adder$ the $multiplier$ and the $decoding\ unit$ is 61.34%, 54.77% and 62.57% respectively.

Figure 4 contains the stress sections taken from the best individuals generated by the algorithm for every module that was considered. Before the execution of each stress section, an initialization sequence takes place which begins with the activation of the $RESET$ signal, which initializes every register with the value 0. Afterwards, a $jump$ instruction is issued that stirs the flow of the program to the stress section. Note, that for the cases of the $adder$ and the $multiplier$ individual, besides the initialization stemming from the activation of the asynchronous $RESET$ signal, an additional initialization phase takes place in order to load the operand values, which will be then used to stress the respective module, to the processor's registers.

In order to compare with our results, we performed a logic simulation of a test program for our processor that reaches 85% fault coverage. For every module of interest, we isolated sequences of instructions of the same size with the generated individuals and measured the switching activity they produced on the modules. Specifically, we evaluated every sequence of instructions by applying the same fitness function which calculates the total stress, induced by the sequence, over the

```
_stress:
/* Setting up the Operands */
    movhi r3,28444
    ori   r3, r3, 45727
    movhi r4, 36228
    ori   r4, r4, 29078
    movhi r5, 35571
    ori   r5, r5, 42079
    movhi r6, 13317
    ori   r6, r6, 8555
    movhi r7, 21419
    ori   r7, r7, 25387
    movhi r8, 53731
    ori   r8, r8, 24000

/* Repeatable Stress Section */
    add  r9, r6, r3
    add  r9, r8, r8
    add  r9, r3, r5
```

(a) Adder

```
_stress:
/* Setting up the Operands */
    movhi r3, 48638
    ori   r3, r3, 60926
    movhi r4, 30328
    ori   r4, r4, 49735
    movhi r5, 54743
    ori   r5, r5, 14163
    movhi r6, 23908
    ori   r6, r6, 37219
    movhi r7, 56247
    ori   r7, r7, 47987
    movhi r8, 14108
    ori   r8, r8, 19615

/* Repeatable Stress Section */
    mul  r9, r6, r7
    mul  r9, r4, r3
    mul  r9, r5, r5
```

(b) Multiplier

```
_stress:
/* Repeatable Stress Section */
    addi r23, r31, -6172
    and  r8, r8, r3
    xori r26, r14, -6554
```

(c) Decoding Unit

Fig. 4: Best Individuals for Every Module

(theoretical) maximum. The results are reported in the lower half of Table II. If we compare with the stress programs generated we can see that they not only induce higher stress on the respective pipeline modules, but also in terms of number of nets being sensitized.

VI. Conclusions

While in most scenarios the minimization of a circuit's switching activity is crucial during the device testing to avoid effects such as overheating, there are cases e.g., during BI, where the goal is the maximization of the switching activity (of the whole CUT or certain sub-modules). During BI the sustained maximization of the system's switching activity via functional stimuli could aid to maximize the stress and thus to screen out early failures.

We proposed an algorithm based on an evolutionary technique able to produce high quality stress programs which consist of short repeatable sequences of instructions. In this paper we focused on the case that the CUT is a fully pipelined processor and showed the effectiveness of the proposed solution in generating stress instruction sequences targeting the $adder$, the $multiplier$ and the $decoding\ unit$. The proposed algorithm can be further applied to more complex processors, given the functional units (sub-modules) of the system whose stress is of interest.

The proposed method does not rely on any kind of dependencies (e.g., pre-existing code). It generates stress sequences from the ground up with acceptable CPU times and implementation complexity. The stress induced generated programs for the $adder$, the $multiplier$ and the $decoding\ unit$ was found to be 61.34%, 54.77% and 62.57%, respectively while also managing to sensitize most of the modules nets.

Work is currently conducted to extend the proposed method by considering the generation of stress programs for more modules (or sub-modules) of the processor.

References

[1] C. He, "Advanced Burn-In - An Optimized Product Stress and Test Flow for Automotive Microcontrollers," in *2019 IEEE International Test Conference (ITC)*. Washington, DC, USA: IEEE, Nov. 2019, pp. 1–6.

[2] R. Cantoro, M. Sonza Reorda, A. Rohani, and H. Kerkhoff, "On the maximization of the sustained switching activity in a processor," in *2015 IEEE 21st International On-Line Testing Symposium (IOLTS)*. Halkidiki, Greece: IEEE, Jul. 2015, pp. 34–35.

[3] Y. Zhang, Z. Peng, J. Jiang, H. Li, and M. Fujita, "Temperature-Aware Software-Based Self-Testing for Delay Faults," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2015*. Grenoble, France: IEEE Conference Publications, 2015, pp. 423–428.

[4] N. Hage, R. Gulve, M. Fujita, and V. Singh, "Instruction-based self-test for delay faults maximizing operating temperature," in *2017 IEEE 23rd International Symposium on On-Line Testing and Robust System Design (IOLTS)*. Thessaloniki, Greece: IEEE, Jul. 2017, pp. 259–264.

[5] I. Polian, J. Anders, S. Becker, P. Bernardi, K. Chakrabarty, N. El-Hamawy, M. Sauer, A. Singh, M. Sonza Reorda, and S. Wagner, "Exploring the Mysteries of System-Level Test," in *Asian Test Symposium (ATS), 2020*. Virtual Conference: IEEE, Nov. 2021.

[6] A. Sagahyroon, "Maximizing heat dissipation for burn-in testing," in *IEEE CCECE2002. Canadian Conference on Electrical and Computer Engineering. Conference Proceedings (Cat. No.02CH37373)*, vol. 1. Winnipeg, Man., Canada: IEEE, 2002, pp. 399–402.

[7] Kuo Chan Huang, Chung Len Lee, and J. Chen, "Maximization of power dissipation under random excitation for burn-in testing," in *Proceedings International Test Conference 1998 (IEEE Cat. No.98CH36270)*. Washington, DC, USA: Int. Test Conference, 1998, pp. 567–576.

[8] F. Aloul and A. Sagahyroon, "Using SAT techniques in dynamic burn-in vector generation," in *2010 15th IEEE Mediterranean Electrotechnical Conference (MELECON)*. Valletta, Malta: IEEE, Jul. 2010, pp. 1448–1452.

[9] N. I. Deligiannis, R. Cantoro, M. Sauer, B. Becker, and M. Sonza Reorda, "New Techniques for the Automatic Identification of Uncontrollable Lines in a CPU Core," in *VLSI Test Symposium (VTS), 2021*. Virtual Conference: IEEE, Apr. 2021.

[10] E. Sanchez, M. Schillaci, and G. Squillero, *Evolutionary Optimization: the μGP toolkit*, 2011th ed. Berlin ; New York: Springer, Apr. 2011.

[11] "OpenRISC," https://openrisc.io, [Online; accessed 15-Apr-2021].

[12] "Silvaco 45nm Open Cell Library," https://si2.org/open-cell-library, [Online; accessed 15-Apr-2021].