2021 International Conference on Electrical, Computer, Communications and Mechatronics Engineering (ICECCME) | 978-1-6654-1262-9/21/$31.00 ©2021 IEEE | DOI: 10.1109/ICECCME52200.2021.9590967

Accelerating Tests of Arithmetic Circuits Through On-FPGA Stimuli Generation and Their Reduction

Jakub Lojda, Jakub Podivinsky, Ondrej Cekan, Zdenek Kotasek
Brno University of Technology, Faculty of Information Technology, Centre of Excellence IT4Innovations
Bozetechova 2, 612 66 Brno, Czech Republic
Email: {ilojda, ipodivinsky, icekan, kotasek}@fit.vutbr.cz

***Abstract*—This paper evaluates the possibility to accelerate fault tolerance evaluation of arithmetic circuits through reduced stimuli. In our research, we used a simplistic on-chip stimuli generator producing numbers in a row with a certain step (i.e. every nth number). The results are obtained through experimentation on a real HW Field Programmable Gate Array. The results confirm the hypothesis, that there might exist appropriate settings, for which the critical bit detection precision becomes only slightly worse but the reliability verification will accelerate significantly. Thus, the correct detection of critical bits in relation to step size is evaluated as certain steps provide significantly lower precision of the estimation than others. Our data show that the steps of sizes larger than 30 do not provide any further effective acceleration. In this paper, evaluations requiring error rate per fault injection are also considered. We also propose a novel stair chart to illustrate the measurement of the error rate per each fault. The results show that the size of the circuit had a minimal impact on the precision. General conclusion is that, by tuning the proper settings of the simplistic generator, significant acceleration of the evaluation can be achieved. The low area overhead of the reduced stimuli generator leaves the saved resources to the tested unit, which in the case of parallel evaluation further supports the acceleration.**

***Index Terms*—Fault-tolerant System Design Evaluation, Fault Tolerance Property Estimation, Functional Verification, High-level Synthesis, Test Bed Generation.**

I. Introduction

Certain types of electronic systems are required to be very durable as their failure might cause financial losses or expose human lives to a risk. Some of these systems must be sufficiently robust simply because they cannot easily be repaired manually, such as space probes. Generally two approaches exist in order to design robust and durable systems: 1) *Fault Avoidance* (FA) [1] which aims to select durable components in order to produce durable system. 2) *Fault Tolerance* (FT) [2] on the other hand accepts the problem of unreliable components and tries to compose a durable system from unreliable components through the system structure modifications.

Moreover, ever-growing demands on electronics increase their design complexity. For this reason, *High-level Synthesis* (HLS) became a common approach to electronic system design. HLS is able to (with certain limitations resulting from the target HW) create *Register-transfer Level* (RTL) description from an algorithm written in a higher programming language (e.g. C, C++, ...). This results in the combination of the FT and HLS approaches with the vision of lowering the time needed to design a system.

FT systems often utilize (but are not limited to) the so-called *Field Programmable Gate Arrays* (FPGAs). FPGAs are often used in space environment to accelerate computation. For example, the *National Aeronautics and Space Administration* (NASA) Perseverance rover [3] utilizes Xilinx FPGAs to accelerate its searching for signs of life on Mars [4]. Such type of use creates a strong motivation to test FPGA-implemented arithmetic components. Quick and accurate evaluation of the component reliability is very important for its designer, whether it is a manually conducted design or automatic design flow. We shorten the evaluation times through HW acceleration – the test controller, including its test pattern generator, is temporarily added to the FPGA technology, very near the tested component (i.e. on the same FPGA). Such approach reduces the bandwidth and for smaller components, parallel evaluation technique can be utilized. As a bonus to this, the tested unit is run at its design speed, outperforming the time complexity of any current simulation approach. The test is usually held with usage of the so-called artificial fault injection and functional verification. In this research paper, we observe the influences of test stimuli generation on the resulting precision and time duration of fault detection. Stimuli generation greatly affects the length of a test. This is very important as it brings a potential to further research and considerably accelerate the evaluation of the designed component. It is also important for computer aided design of fault-tolerant systems, which is also our topic of research. So far, the evaluation was the most time-consuming part of our method run time. The goal of this research is to reduce stimuli amount in order accelerate the evaluation of a design.

Various approaches to test data generation are published in the literature. These are, however, in opposition targeting the generation in a simulated environment. The authors of [5] show a concept of efficient stimuli generation based on constraint random verification techniques. The same authors extend their research with the coverage-guided sampling in their paper [6]. In paper [7], another constrained random stimuli generation method is presented. The authors of the paper use genetic algorithm to provide fast test coverage, also a case study on the PCI Express component is shown. The authors of [8] developed a new framework that generates stimuli for parallel VHDL processes. In [9], the authors show a new approach to guide fault injection in order to ensure representativeness of the obtained data. Also the authors utilize data mining approaches in their research. In [10], a *Universal Verification Methodology* (UVM) implemented in SystemVerilog is shown. The authors of the paper focus on

the *Joint Test Action Group* (JTAG) interface.

This paper is organized as follows: Section II puts this research in context to our automatic test bed generation framework. Section III presents the two main approaches to on-FPGA stimuli generation. The experimental setup is proposed in Section IV. The case study and experimental results are presented in Section V. Section VI concludes the paper.

II. Test Bed Generation Framework

We developed a framework [11] that produces synthesizable test beds which run functional verification directly on an FPGA, which we utilized in research papers [12], [13] and further extended in [14]. After the verification started, an artificial fault is injected directly into the tested component and the changes in behavior of its outputs are compared to the so-called golden (i.e. reference) unit and further tracked. It is thus possible to verify fault tolerance properties of the designed component, i.e. the *Unit Under Test* (UUT) on a real HW. The test bed includes stimuli generator that operates directly on the FPGA in order to maximize throughput and allow the at-speed testing. Fault tolerance of an FPGA circuit design is expressed by the cardinality of the critical bit set. Critical bit (also called sensitive bit) is a bit the flip of which is observable on the design output pins, as it causes change in output data for a given test. It is obvious that the detection of each of the critical bits is costly, as the number of tests (i.e. functional verification runs) is high to identify each critical bit.

The block diagram of the test bed generated using our framework is shown in Fig. 1. The scenario of each test is managed by the *Finite State Machine* (FSM) inside the *eXperiment Control Unit* (XCU). The *Input Generation Unit* (IGU) generates input stimuli which are directed to tested units (i.e. UUTs) in the Unit Instantiation Area. The *Output Compare Unit* (OCU) compares the results between the so-called golden unit (i.e. reference unit) and UUTs to which a fault was injected. The Failure Capture Unit then stores the number of mismatching output transactions into the Register File which is accessible to the control PC with the usage of the ICON [15] and VIO [16] IP Cores and the SW ChipScope Engine TCL Interface [17]. The faults are injected with the usage of our Fault Injector [18] which is able to inject permanent faults into utilized *Look-up Table* (LUT) contents.

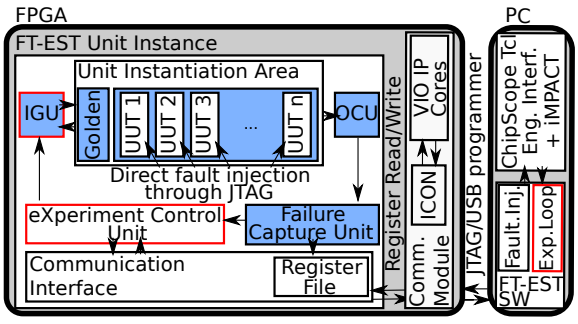

Fig. 1. Block diagram of automatically generated test bed; HW (FPGA) part on the left, SW (PC) part on the right.

The majority of the test bed is generated automatically and does not require a modification, however, three parts that are marked by red color in the Fig. 1 might be adapted to suit the UUT in its particular test requirements. These parts include XCU, which manages the test scenario and the IGU, which generates the test stimuli. Also the Experiment Loop must be prepared to suit the test scenario. This includes the setup of the required parameters for autonomous verification execution and the fault injection strategy, which injects faults into the bitstream configuration string using our Fault Injector [18].

III. On-chip Stimuli Generators

Generally, an on-chip stimuli generator can be implemented in two contradictory manners: 1) Relatively complex generator with very good setting possibilities and a high test coverage. Such generator would however occupy a relatively large FPGA area which reduces the remaining FPGA space left for UUTs, thus lowering the number of parallel instances that can be tested at the same time on one FPGA. 2) The second approach is to create a very specific, however simplistic generator which saves space on the FPGA and allows to increase the number of simultaneously tested units for our future (i.e. real and parallel) tests. The parallel testing of units is possible for scenarios, in which multiple random faults are accumulating during one run into one unit. The possibilities to adjust the coverage of such solution are then limited and correspond to the simplistic nature of such generator. The flow of preparation and usage of HW stimuli generator can be seen in Fig. 2.

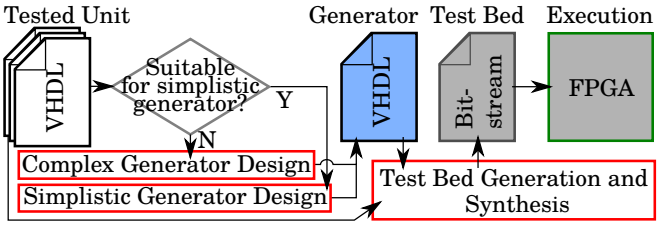

Fig. 2. The flow of preparation and usage of a stimuli generator.

In this section, both possible approaches to on-chip stimuli generation are presented. However, the approach finally utilized in our study is described in the last part of this section.

A. Complex Generators

For complex systems or for sophisticated stimuli generation, manual IGU creation at the HDL description level is very difficult, therefore, in the following text, the advanced way to generate the content of the IGU from the description of formal grammar will be presented to illustrate its design process.

The main idea is to design a particular grammar which can be easily transformed into adequate FSM. Tools that can transform an FSM directly to VHDL description (e.g. Kiss2-to-VHDL [19]) exist. Such description can be directly analyzed and synthesized. Grammars and FSMs are mutually transferable which can be turned to a large benefit. In our previous research, we developed the *Universal Stimuli Generator* (USG) [20] which is based on the formal grammar, and therefore it can be used for the purposes of an FSM generation.

Generally, stimuli generation for complex systems is based on solving the *Constraint Satisfaction Problem* (CSP) [21]. It is a mathematical problem which is based on assigning values from certain domain into variables such that all specified constraints are respected. For the purpose of stimuli generation for various systems, we combined CSP with formal grammar.

The USG is based on our *Probabilistic Constrained Grammar* (PCG) which is based on two input structures, through which input stimuli can be both described and generated. The first structure defines the probabilistic context-free grammar and the second structure defines the constraints for this grammar. Both input structures form the newly defined grammar (PCG) which is the pair G:

`G = (H,C); where:`

H is a probabilistic context-free grammar.
C is a ordered list of constraints for the grammar H.

The probabilistic context-free grammar (PCFG) [22] is the 5-tuple:

`H = (N,T,R,S,P); where:`

N is a finite set of non-terminal symbols.
T is a finite set of terminal symbols, N∩T = 0.
R is a finite set of production rules with form A $\rightarrow \alpha$, where A $\in$ N a $\alpha \in$ (N∪T)*.
S is the starting non-terminal.
P is a finite set of probabilities for production rules.

The PCFG is the common context-free grammar into which the probability values for production rules were delivered. Certain production rule may have adjusted its probability which causes an increased or decreased chance for rewriting. The defined probability value can be modified by the constraint during the generation process. It ensures stimuli generation through the selection of appropriate production rules.

The constraint is the 5-tuple:

`C = (`R_S`,`R_D`,P,[`R_E`],[O]); where:`

$\mathbf{R}_S$ is the production rule which invoke this constraint.
$\mathbf{R}_D$ is the production rule for which the probability is changed.
P is the new probability value.
$\mathbf{R}_E$ (optional) is the production rule which application causes the abolition of the set probability.
O (optional) is the count of R_E applications before abolition of the constraint.

The constraint in PCG definition performs one of the possible solutions for solving the CSP which is a technique called *propagating constraints*. In this formal grammar, the propagating constraints limit the domain of production rules for each non-terminal symbol such that only certain rules are activated. Thanks to that, it can generate valid stimuli in terms of correct syntax and especially to keep the correctness of semantics which is a necessity.

One of the challenges for generating more complex stimuli defined in this way is to describe and generate FSM. For this purpose, the above mentioned tool Kiss2-to-VHDL can be used. Kiss2-to-VHDL uses its specific format of input description called *Kiss2 code* which can be transformed into equivalent FSM in VHDL description through this tool. The PCG has to be defined in such format to setup the USG to generate the required Kiss2 code. The generated Kiss2 code then represents the required stimuli. The Kiss2 code is a plain text which encodes information about inputs, outputs, transitions and states of the required FSM. This format must be understood in detail and encoded into our definition of the formal grammar. Fig. 3 shows the block schema of this transformation process. Through this, it is possible to describe the input stimuli on the higher abstraction level to generate VHDL code directly. With bit of modifications of generator inputs, we should be able to generate different variants of the IGU to obtain the best stimulus properties and the highest coverage of key system functions.

B. Simplistic Generators

The second and opposite approach to the IGU design is to simplify it, resulting in a smaller chip area which belongs to the test bed overhead. There are several possibilities to create a simple, yet effective stimuli generator. Data-oriented UUTs require a production of a large amount of data to test them. Related generators include, but are not limited to, a sequential generation of all binary combinations which can be easily implemented as a digital counter; or a pseudo-random number generation which can be implemented as a linear-feedback shift register, in order to save space. Certain generators are not intended to generate a large amount of data, but, in opposite, to generate signals to verify a function of, for example, an electronic controller. Such generators can be appropriately implemented as an FSM. For all these cases, simplistic design of a generator can significantly save the FPGA area. Also, it lowers the requirements to prepare a specific single-purpose stimuli generator. This must be considered, as the preparation of a specific generator requires its separate design process in order to develop and implement the generator. Simplistic generation is also usually easily scalable, as it requires only minor modifications to alter its output bit width. In our research, we investigate the possibility to use a counter based simplistic generator in order to process all the possible combinations on the inputs of a tested unit. The manual design flow of a simplistic generator is displayed on Fig. 4.

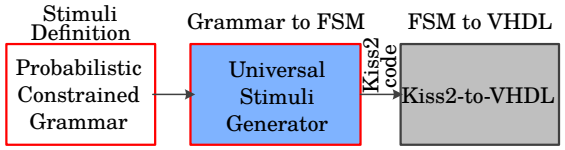

Fig. 3. The transformation of the probabilistic constrained grammar into its FSM through the USG and then into the VHDL description through the Kiss2-to-VHDL tool.

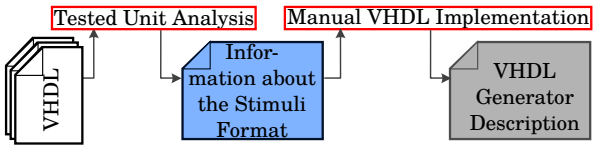

Fig. 4. The manual design flow of a simplistic stimuli generator.

This research focuses on testing a special class of circuits – the arithmetic circuits. Such circuits must be properly tested, as their failure causes incorrect results output. We use a counter to produce stimuli for our benchmark circuits. The counter has a possibility to set the starting value START and the target value END. Also, it is able to count between these values with a specified STEP. It is obvious, that with increasing the value STEP, the stimuli are evenly reduced. This way, also the test run time reduces and the complete evaluation accelerates. If we intend to estimate the fault tolerance of the UUT more precisely which is always welcome, it is good to perform as many tests as possible. This is why it is desirable to minimize the time that is needed by each test, allowing us to make more tests. This brings us to the requirement to tune the generator parameters in order to minimize the test run time length. This offers a hypothesis, that there might exist appropriate settings for the simplistic generator, for which the critical bit detection precision becomes only slightly worse but the test will accelerate significantly.

IV. The Experimental Setup

For the case study experiments in this paper, we use our automatic test bed generation framework in conjunction with a simplistic approach to generate stimuli directly on the FPGA for at-speed testing. As the generator, we use a HW implementation of a counter that is SW-adjustable from the PC. We investigate the possibility to maximize its critical bit

detection probability while shortening the run time of one test. The usage of a simplistic stimuli generator in our research is motivated by the effort to save the most of the FPGA area for the UUTs. This is because for certain test scenarios, it is possible to significantly accelerate the test or refine the results through the multiple unit instantiation, and thus, the saved space brings the possibility to further accelerate the test.

The resources utilized by the test bed components (i.e. the area overhead that is necessary to hold the autonomous on-chip test) are shown in Fig. 5. We use the Xilinx Virtex 5 technology in our tests. The chart shows (a) the slices; (b) LUTs; and (c) registers occupied by the test bed logic. As can be seen, the test bed logic components (i.e. without the UUT) occupy only 6.13 % of the FPGA, leaving the remaining 93 % to instantiate the UUTs, as can be seen in the part (a). The simplistic counter-based IGU requires nearly 18 % of LUTs of the complete test bed logic, as can be seen in (b). This is because in its nature, the IGU adds numbers which must be implemented as an adder. Although for the powers of two, the IGU can be implemented as a bit-shift operation, this is not our case, as the IGU is configurable from the PC, thus its design remains constant among the step-size configurations. On the other hand, the remaining part of the test bed logic does not occupy a large area of LUTs. Also, as can be seen in (c), the IGU does not require a lot of registers, as the input parameter values are stored and connected from the register file and only the counter register is implemented within the IGU. It is important to note that keeping the occupied space as low as possible leaves more space for the instances of UUTs, allowing further acceleration through massive parallel evaluation.

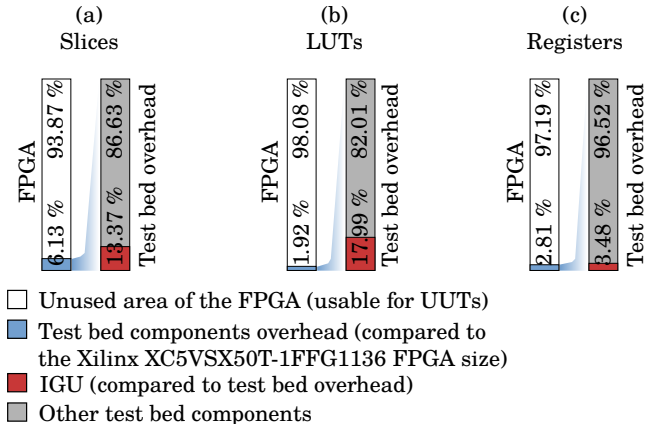

Fig. 5. Virtex 5 FPGA resources used by the test bed components (overhead) and proportionally by the IGU; the remaining free space can be used to instantiate UUTs.

In order to test the behavior for unhardened and fault-tolerant designs as well, we utilize the *Redundant Data Types* (RDTs) [23] technique for HLS. This method of treatment is based on the modification of data types in the source code implementation. In our research, we use the C++ programming language, as it supports the so-called C++ Templates [24] to easily decorate C++ native data types and modify operations associated with them. The new data types can be used identically to the native C++ data types, however, inside they extend the computation checks, add temporal or spatial redundancy, etc. For example, a *Triple Modular Redundancy* (TMR) template for a new data-type set stores each data structure three times. Each operation with such data structure is also triplicated. In order to use the RDT approach, only the data type names must be decorated with the particular C++ template and after the synthesis, the resulting circuit is hardened. Moreover, the RDT technique allows to implement transition mechanisms between different fault tolerance structures (e.g. TMR, Duplication with Comparison, etc.) also implemented as RDTs. This allows to combine various fault tolerance approaches in one circuit.

V. The Case Study and Experimental Results

For testing purposes, we selected six various benchmark circuits differing in their complexity and fault tolerance. More complex combinational circuits can be partitioned into smaller slices before their testing. Our experimental circuits include: 1) addition – a simple adder for two 16-bit unsigned integers; 2) addition_triple – similar to (1), however RDT triple incorporating TMR to the data path was used; 3) crc8 – calculates the *Cyclic Redundancy Check* (CRC) from 32 bits to an 8 bit long checksum; 4) crc8_triple – similar to (3), however all data structures were hardened using the RDT triple approach. 5) composed – a circuit that calculates the number of high-set bits in an addition of two 16-bit numbers and one constant number; and 6) composed_triple – similar to (5), however triple RDT was applied to each data structure. The circuits were originally written in the C++ language and the fault-tolerant units were created from their unhardened counterparts after a voter was included to each of the final output pins of each circuit. Circuits were synthesized using Mentor Graphics Catapult C *University Version* (UV) 8.2b [25] to obtain their RTL description. These were then settled into the environment of the test bed generation framework and the complete test bed was synthesized using the Xilinx *Integrated Synthesis Environment* (ISE) 14.7 [26]. Xilinx ML506 boards [27] are used in our case study. Parameters of synthesized benchmarks are shown in Table I.

TABLE I
Parameters of Synthesized Benchmarks

UUT Name	LUT size [b]	Critical bits [b]	Critical bits [%]	Input Pins [-]	Output Pins [-]
addition	4288 b	163 b	3.8 %	32	16
addition_triple	8320 b	162 b	1.95 %	32	16
crc8	4800 b	977 b	20.35 %	32	8
crc8_triple	6592 b	819 b	12.42 %	32	8
composed	9120 b	1521 b	16.68 %	32	13
composed_triple	19648 b	1056 b	5.37 %	32	13

As each of the benchmarks has 32 bit wide input, the stimuli of one test were from the interval of 0 to $2^{32} - 1$. For each test, exactly one bit from the selected part of the bitstream was permanently flipped during the whole execution of the test. Such test was performed for each of the selected bits. One or more incorrect outputs during the one test then indicate the sensitivity of the currently tested bit. This is how a complete list of sensitive bits is obtained.

A. Test Length and Detection Success Rate

The previously described process of experiments utilized the step of size 1, to obtain accurate list of critical bits. This process was subsequently repeated for stimuli generated with the step size 2, 3, ... 200. It is obvious that the test (i.e. one functional verification) run length decreases with increasing step size. As can be seen in Fig. 6, with increasing the step size, the verification accelerates. Constant and dynamic part of time consumed for one test can be identified in the chart. The constant part of the time is connected with the communication, UUT preparation and fault injection overhead. The dynamic

part depends on the number of test stimuli that the UUT has to process. The chart illustrates that near the step of size 30, the acceleration ceases to be beneficial, considering the time consumption of the UUT refresh and fault injection which are not dependent on the number of stimuli transactions.

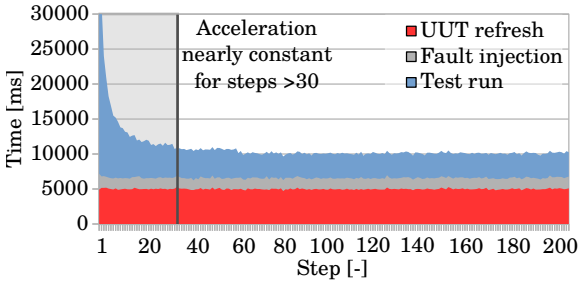

Fig. 6. Time (in milliseconds) consumed by various stages of one test for stimuli step sizes of 1 – 200.

For step sizes higher than 1, the resulting percentage of critical bits is literally an estimation. This is the consequence of the lower coverage, which might render certain critical bits undetectable. As can be seen in Fig. 7, higher steps lower the precision of such estimation. But also the even step sizes and especially the powers of two cause significantly higher inaccuracy of the estimation. Such step sizes do not cause changes in certain least significant bits of the stimuli transactions, thus decreasing the precision significantly.

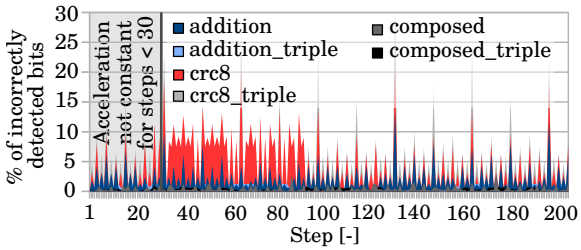

Fig. 7. Percentage of undetected critical bits for stimuli step sizes of 1 – 200.

If for each step size a sum from all the six inaccuracy measurements is taken, then the best (i.e. the lowest) value is 1.37 percentage points belonging to the step of size 91.

B. Step Size vs. Error Rate

So far in this paper, we have been considering only detection of a critical bit. However, an advanced evaluation might consider error rate that a particular bit-flip causes (i.e. the number or percentage of incorrect result transactions per test for one bit-flip). With such information, the failures (e.g. critical bits) can be categorized by their severity. One test produces the number of incorrect results which is nonzero for critical bits. Taking a test for each selected bit of the bitstream, a multiset of error rates is produced. Such multiset can be sorted and visualized in a chart we call the *stair chart* in our work. The bitstream addresses then correspond to the x axis and error rate percentage belongs to the y axis. For the purposes of visualization, the x axis can be marked by the measure of bits as a dimension rather than their addresses. Example of such stair chart for the addition UUT can be seen in Fig. 8.

A set of 200 charts is obtained for 200 experiments of different stimuli step sizes per each UUT. These chart sets can be visualized and studied after a *stair chart heat map* is created from them. If we suppose the step size does not influence the shape of the stair chart, the heat map color would

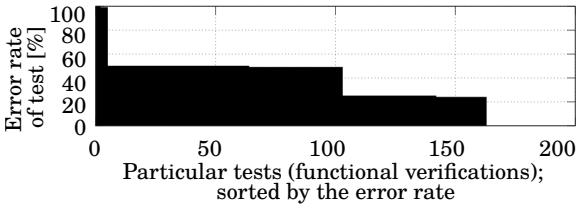

Fig. 8. Sorted error rates for the addition UUT as an example of the stair chart for stimuli step equal to 1.

then sharply follow its shape and no color gradients would be observed. In practice, the step size influences the stair chart shape. The marginal cases show up as a lighter shade of color on the charts. Fig. 9 shows the heat maps for each of the benchmark circuits. Considering the stair chart shape, the step size generally influences only slightly the error rate measurement. As well for various sizes of the benchmarks, the results indicate that the design size influences the shape by adding stairs. However, the design size affects the accuracy only slightly, as can be seen on the color scale. Also, an interesting fact concerning circuit evaluation can be observed: larger and more complex designs have more stairs on the chart. We believe that this is actually indicating the growth of the set of potential failure types that can affect the design. The general conclusion is that for the applications requiring to measure the error rate per each fault injection, the step size has a negligible impact on the accuracy while one can still profit from the acceleration, resulting in nearly 5 times smaller test run length for steps of sizes larger than 30.

VI. Conclusions

In this paper, the possibility to accelerate fault tolerance evaluation through reduced stimuli set was examined. In our research, a simple on-chip stimuli generator was used to produce numbers in a row with a certain step. Such simple generator saves space on the FPGA. In the case study, six different circuit designs were evaluated. At first, the attention was paid to the acceleration. The results show that the steps of sizes larger than 30 do not provide any further effective acceleration considering the constant fixed run times of UUT reconfiguration and fault injection. Secondly, the paper focuses on the correct detection of critical bits as some steps provide significantly lower precision of the estimation than others. In the third part of the paper, evaluations requiring error rate per fault injections are considered. The results show that the size of the circuit has a minimal impact on the precision. The results presented in the case study are useful especially for combinational circuits. However, generally, this research is useful for everybody utilizing on-chip testing and, also particularly, for our research of fault-tolerant system design automation, in which the speed of evaluation determines the size of state space our tool is able to explore.

Acknowledgments

This work was supported by the Brno University of Technology under number FIT-S-20-6309.

References

[1] J.-C. Geffroy and G. Motet, *Design of Dependable Computing Systems*. Kluwer Academic Publishers, 2002.

[2] I. Koren and C. M. Krishna, *Fault-Tolerant Systems*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[3] NASA, "Mars 2020 Perseverance Rover," 2020, accessed: 2021-04-11. [Online]. Available: https://mars.nasa.gov/mars2020/

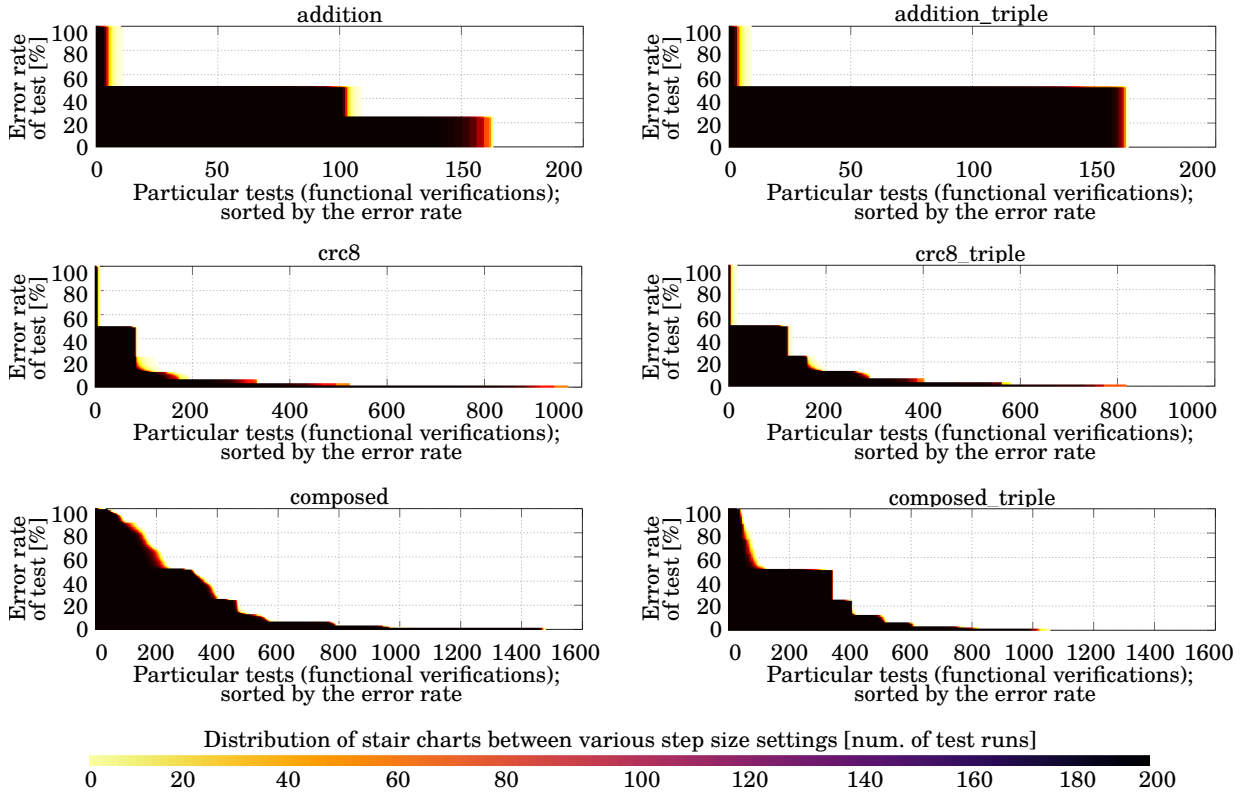

Fig. 9. Heat maps created from the stair charts of each benchmark circuit; black color indicates the highest representation while the yellow color indicates the lowest representation.

[4] Farhad Fallahlalehzari, "How does the Mars Perseverance rover benefit from FPGAs as the main processing units?" accessed: 2021-04-11. [Online]. Available: https://www.aldec.com/en/company/blog/188–how-does-the-mars-perseverance-rover-benefit-from-fpgas-as-the-main-processing-units

[5] R. Dutra, J. Bachrach, and K. Sen, "SMTSampler: Efficient Stimulus Generation from Complex SMT Constraints," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov 2018, pp. 1–8.

[6] R. Dutra, J. Bachrach, and K. Sen, "GUIDEDSAMPLER: Coverage-guided Sampling of SMT Solutions," in *2019 Formal Methods in Computer Aided Design (FMCAD)*, Oct 2019, pp. 203–211.

[7] W. Jiawen, L. Zhigui, W. Suliang, L. Yang, L. Yufei, and Y. Hao, "Coverage-directed Stimulus Generation Using a Genetic Algorithm," in *2013 International SoC Design Conference (ISOCC)*, Nov 2013, pp. 298–301.

[8] V. Jusas and T. Neverdauskas, "Stimuli Generator for Testing Processes in VHDL," in *2014 NORCHIP*, Oct 2014, pp. 1–4.

[9] F. Cerveira, I. Kocsis, R. Barbosa, H. Madeira, and A. Pataricza, "Exploratory data analysis of fault injection campaigns," in *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, 2018, pp. 191–202.

[10] C. Elakkiya, N. S. Murty, C. Babu, and G. Jalan, "Functional Coverage – Driven UVM Based JTAG Verification," in *2017 IEEE International Conference on Computational Intelligence and Computing Research (ICCIC)*, Dec 2017, pp. 1–7.

[11] J. Lojda, J. Podivinsky, O. Cekan, R. Panek, and Z. Kotasek, "FT-EST Framework: Reliability Estimation for the Purposes of Fault-Tolerant System Design Automation," in *2018 21st Euromicro Conference on Digital System Design (DSD)*, Aug 2018, pp. 244–251.

[12] J. Lojda, J. Podivinsky, O. Cekan, R. Panek, M. Krcma, and Z. Kotasek, "Automatic Design of Reliable Systems Based on the Multiple-choice Knapsack Problem," in *2020 23rd International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, 2020.

[13] J. Lojda, R. Panek, and Z. Kotasek, "Automatically-Designed Fault-Tolerant Systems: Failed Partitions Recovery," in *Accepted for Presentation on: 2021 IEEE East-West Design and Test Symposium (EWDTS), Batumi, Georgia*, Sep 2021.

[14] J. Lojda, R. Panek, and Z. Kotasek, "Automatic Design of Fault-Tolerant Systems for VHDL and SRAM-based FPGAs," in *Accepted for Presentation on: 2021 24th Euromicro Conference on Digital System Design (DSD), Palermo, Sicily*, Sep 2021.

[15] Xilinx Inc., "LogiCORE IP ChipScope Pro Integrated Controller (ICON) Documentation," https://www.xilinx.com/support/documentation/ip_documentation/chipscope_icon/v1_05_a/chipscope_icon.pdf, Jun. 2011, accessed: 2018-02-15.

[16] Xilinx Inc., "ChipScope Pro VIO Documentation," https://www.xilinx.com/support/documentation/ip_documentation/chipscope_vio.pdf, Sep. 2009, accessed: 2018-02-15.

[17] Xilinx Inc., "ChipScope Pro 11.4 Software and Cores User Guide," https://www.xilinx.com/support/documentation/sw_manuals/xilinx11/chipscope_pro_sw_cores_ug029.pdf, Dec. 2009, accessed: 2018-02-15.

[18] M. Straka, J. Kastil, and Z. Kotasek, "SEU Simulation Framework for Xilinx FPGA: First Step Towards Testing Fault Tolerant Systems," in *14th EUROMICRO Conference on Digital System Design*. IEEE Computer Society, 2011, pp. 223–230.

[19] A. Abdel-Hamid, M. Zaki, and S. Tahar, "A Tool Converting Finite State Machine to VHDL," *Canadian Conference on Electrical and Computer Engineering*, vol. 4, pp. 1907 – 1910 Vol.4, 06 2004.

[20] O. Cekan, R. Panek, and Z. Kotasek, "Input and Output Generation for the Verification of ALU: A Use Case," in *2018 IEEE East-West Design & Test Symposium (EWDTS)*. IEEE, 2018, pp. 1–6.

[21] V. Kumar, "Algorithms for constraint satisfaction problems: A survey," *AI MAGAZINE*, vol. 13, no. 1, pp. 32–44, 1992.

[22] R. Giegerich, *Introduction to Stochastic Context Free Grammars*, J. Gorodkin and L. W. Ruzzo, Eds. Totowa, NJ: Humana Press, 2014.

[23] J. Lojda, J. Podivinsky, Z. Kotasek, and M. Krcma, "Data Types and Operations Modifications: A Practical Approach to Fault Tolerance in HLS," in *2017 IEEE East-West Design Test Symposium (EWDTS)*, Sept 2017, pp. 1–6.

[24] D. Vandevoorde and N. M. Josuttis, *C++ Templates*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[25] M. Graphics, "Catapult HLS," https://www.mentor.com/hls-lp/catapult-high-level-synthesis/, 2017, accessed: 2017-07-07.

[26] Xilinx Inc., "ISE Design Suite," https://www.xilinx.com/products/design-tools/ise-design-suite.html, 2017, accessed: 2017-07-07.

[27] Xilinx Inc., "Ml506 Evaluation Platform User Guide," *UG347 (v3. 1.2)*, 2011.