Software Product Reliability Based on Basic Block Metrics Recomposition

Tiziano Fiorucci, Giorgio Di Natale, Jean-Marc Daveau, Philippe Roche

Software Product Reliability Based on Basic Block Metrics Recomposition

Tiziano Fiorucci[1,2], Giorgio Di Natale[2], Jean-Marc Daveau[1], Philippe Roche[1]

[1]STMicroelectronics, 850 Rue Jean Monnet 38926 Crolles Cedex, France
[2]Univ. Grenoble Alpes, CNRS, Grenoble INP*, TIMA, 38000 Grenoble, France

***Abstract*—In the context of functional verification, the focus has always been on hardware and its ability to be both resilient to errors and to recover from them autonomously. In order to evaluate these characteristics, an extensive use of Fault Injection tools is made to achieve clear and granular results. These testing campaigns are carried out on the entire DUT and require a consistent amount of time and computational resources. The possibility of reducing these costs applying modern techniques as the study of the Dysfunctional State Machine or the proof of concept regarding the composability of single block fault injection campaigns to obtain a library of component of which the reliability metrics are well known, as already been extensively discussed and proven on hardware. In this work instead the application of this methodologies to software is presented for the first time. In order to do so, the software has been divided into basic block, atomic chunks of code having precise carachteristics that will ensure the possibility to study them singularly and then recompose them into a software product which reliability metrics are known, without the need for complete Fault injection campaign.**

I. Introduction

In the last years the density of integration in VLSI systems and microprocessors performances have continuously increased, thanks to the relentless technology scaling. Even though this trend can only continue on its path, several constraints may obstruct the way (power, energy, performance), in particular *reliability* (or cross-layer resilience) can become the more relevant. Hardware redundancy can be used to manage errors at the hardware architecture layer, and eventually even software implemented error detection and correction mechanisms can manage those errors that escalated from the lower layers of the stack [1] [2]. Overall, the goal is to determine the resilience of a particular system in determined conditions, meeting the requirements considering its sensitivity to hardware faults.

It is also true that software failures are not only caused by software implemented faults, as it has been shown [3] the propagation of hardware faults plays a central role, eventually catastrophic. Base on what literature reports on hardware faults evaluation reports [4] [5] it is possible to observe that the percentage of software failure that are caused by pure hardware faults average around 10% [6]. The most famous example is surely the crash of the Mars Polar Lander [7], which cause was established to be dependant on hardware faults resulting in software failure. In that case the lander was not able to settle the legs into their deployed position, which is an hardware fault, and the software gave a wrong order to turn off the engines in the air of Mars, which is a software fault. The system crashed and the entire mission failed.

This paper not only wants to furthermore analyse the behaviour of software failure due to hardware propagated fault but parallely to the main research [8] path that applies these new methodology to Hardware design in order to simplify the reliability assessment, the idea of applying the same method in the scope of the assessment of the reliability of software has never been tested. In order to do so, there is the need to specify the main characteristic that Software Products have, fundamental to lay the basis for the described work. Every software can be divided into basic block, atomic chunks of software having the following proprieties:

- One entry point, meaning no code within it is the destination of a jump instruction anywhere in the program.
- One exit point, meaning only the last instruction can cause the program to begin executing code in a different basic block.

Under these circumstances, whenever the first instruction in a basic block is executed, the rest of the instructions are necessarily executed exactly once, in order. The code may be source code, assembly code, or some other sequence of instructions. More formally, a sequence of instructions forms a basic block if:

- The instruction in each position dominates, or always executes before, all those in later positions.
- No other instruction executes between two instructions in the sequence. This definition is more general than the intuitive one in some ways. For example, it allows unconditional jumps to labels not targeted by other jumps. This definition embodies the properties that make basic blocks easy to work with when constructing an algorithm.

The blocks to which control may transfer after reaching the end of a block are called that block's successors, while the blocks from which control may have come when entering a block are called that block's predecessors. The start of a basic block may be jumped to from more than one location. Laid these basis, if, as we'll show in this paper, the reliability metrics extracted for each basic block can be recomposed just knowing the sequence of block required to execute a precise operation, the need for a fault injection campaign on the entire software product doesn't stand anymore.

This paper is organised as follows: the current state of the art is summarised in section II; section III describes the proposed methodology, including its setup, the fault injection procedure and the re-composition of the results from each basic block; a test case is provided in Section IV, while Section V presents the obtained results and sketches some perspectives.

II. State of the Art

The rush to develop a methodology to assess the reliability and availability of electronic systems has speed up together with the increasing complexity of the microelectronic systems and the miniaturization of such devices. In particular an eye has been keep onto the propagation of faults throughout the entire stack of layers that compose the system as whole, starting from the technological layer all the way up to the software/application layer passing through hardware. In particular the extraction

*Institut National Polytechnique Grenoble Alpes

of reliability metrics for software has been the focus of a consistent thread of research [6] [9] [10] that aimed to verify:

1) whether the software respects the specification requirements,
2) the improvement of the software quality and,
3) *the reliability of the software*

Tools to verify the reliability of software, defined as the probability of the correct software performances for specific period of time on specific environments, have been already developed. In particular the SyRA [11] Cross-Layer Soft Error Resilience evaluation framework proposes a solid method to move from the industrial level Cross-Layer evaluation techniques that are still mainly guided by the sole experience of the designers [1]. These methods are all based on the use of fault injection tools, and they all produce satisfying results in their fields. Nevertheless they have limitation, the description of the Software Fault Models have always been based on the simulation of propagation from the hardware architecture up to software routines, assessing their impact in the correctness of the computation as in [12] [13] [14]. Moreover no attention has been given to the enormous effort that this type of campaign require, in terms of time, licences for tools and computational power, for an assessment that is limited to the hardware the application is running on and most importantly on the inputs the software receives to perform its calculation. This makes the assessment completely not re-usable in the future requiring a completely new set of campaigns.

Here the focus will be, instead, put on how the software computation reacts to the vulnerable hardware underneath and most importantly to the de development of a methodology like there are no other example in the related research, the possibility of decomposing the software products to abstract the single basic blocks and perform a reliability assessment on the single, apparently meaningless blocks to then recompose them obtaining the reliability assessment with a huge time and computational power advantage with respect to the existing methods.

III. Methodology

The Classical reliability assesment of Hardware as well as Software is Fault Injection driven. The extensive usage of commercial fault injection tools like the ones provided by **Cadence** [15] or **Synopsys** [16] guarantees the proper exploration of the behaviour of the DUT when subject to SEU or other types of faults. This allows the Verification Engineers to have an idea of the behavior of their design without the need to move onto practical testing in radiation environments, which require a dedicate setup [17] and an expensive and not widely available infrastructure.

These advantages come at two main costs, *time* and *Computational Power*, which are comsumed in great quantities by the above mentioned simulators. Attempts of Optimization and Parallelization have been put in practice before, but they are not tackling the bigger overhead that we need to take care of every time we simulate a design. Let us assume that, as shown in Fig:3 there is the need to test and entire Software Product composed of n basic blocks, this simulation will last as long as the time to initialize T_{init} plus the time of the checker/footer to be executed T_{foot} plus the sum of the duration of all basic blocks multiplied by their multiplicity through the program $m_n \cdot T_{bb_n}$. All multiplied by the number of runs that the simulator has to perform to achieve the desired number of injections I, resulting in:

$$T_{campaign} = I \cdot \left[T_{init} + T_{foot} + \sum_{0}^{N} m_n \cdot T_{bb_n} \right] \quad (1)$$

In which the entire program is executed every time entirely, the method proposed by this paper consist in a fragmented study of the basic blocks composing the software, extracting the same metrics that would be extracted by the same fault injection campaing on the whole Software. In this case, in the same way we did before, it is possible to calculate the time needed to carry out the fault injection campaign as we have defined it now, on separate basic blocks, each of them having their random initialization and checker to ensure functionality.

$$I \cdot \left[T_{init} + T_{foot} + \sum_{0}^{N} T_{bb_n} \right] \quad (2)$$

In this way we have drastically reduced the amount of time needed to perform the same amount of fault injections, just focusing on the single blocks. Moreover, the difference between the two previously calculated timings, will give us the benefit of studying the blocks singularly, as follow:

$$\begin{aligned} I \cdot \left[\sum_{0}^{N} m_n \cdot T_{bb_n} \right] - I \cdot \sum_{0}^{N} T_{bb_n} = \\ = I \cdot \left[\sum_{0}^{N} m_n T_{bb_n} - \sum_{0}^{N} T_{bb_n} \right] = \\ = I \cdot \sum_{0}^{N} T_{bb_n} \cdot (m_n - 1) \end{aligned} \quad (3)$$

which means that we save the time needed for the execution of each basic block multiplied by its multiplicity, minus one that we still have to execute. Clearly this saved time increases with the length of the Software and therefore the multiplicity of the blocks. In particular, the length of the Fault injection campaign on the entire software is linear with respect to the increasing of multiplicity of the basic blocks, for example due to a larger data input, whereas the solution proposed in this paper is linear with respect to the overall number of unique basic blocks, which remain the same regardless of the data.

A. Setup

The first step towards the application of the method described in the previous section, is the identification and of the different basic block that compose the Software Product under analysis. This can be easily carried out automatically by a simple parser. Basic Block at Assembly level are easy to identify and parse thanks to their intrinsic definition of linear chunks of code. It is therefore trivial to identify in the code all those instructions that modify the flow of the program, tearing down the hypothesis of linearity that defines the blocks themselves. For instance, all the jumping and branching point define the end of a block, as well as the beginning of the following one. Labels in the code also identify starting point of basic block, as they are frequently arrival points for the above mentioned jump and branch operations.

```
addi  sp,sp,-48       \\Beginning of BB_1
sw  s0,44(sp)
addi  s0,sp,48
li  a5,3
sw  a5,-48(s0)
li  a5,5
sw  a5,-44(s0)
li  a5,1
sw  a5,-40(s0)
li  a5,2
sw  a5,-36(s0)
li  a5,4
sw  a5,-32(s0)
sw  zero,-20(s0)
```

```
15 j .L2                    \\End of BB_1
16 .L6:                       \\Beginning of BB_2
17 sw  zero,-24(s0)
18 j .L3                    \\End of BB_2
19 .L5:                       \\Beginning of BB_3
20 lw  a5,-24(s0)
21 slli  a5,a5,2
22 addi  a4,s0,-16
23 add a5,a4,a5
24 lw  a4,-32(a5)
25 lw  a5,-24(s0)
26 addi  a5,a5,1
27 slli  a5,a5,2
28 addi  a3,s0,-16
29 add a5,a3,a5
30 lw  a5,-32(a5)
31 bge a5,a4,.L4            \\End of BB3
```

Listing 1: Example of software

For example, the chunk of code reported in [listing1] Includes 3 basic blocks as divided in the comments, having as extremes the jumping/branching operations as well as lables.

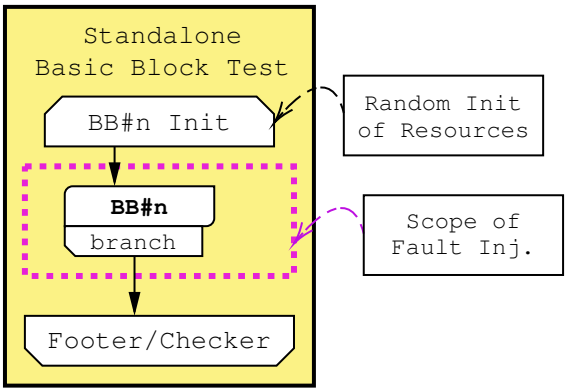

Fig. 1: Block Diagram of the Entire SW Product

Although having the set of basic blocks divided in single file may seem sufficient, there still the need to initialize all the resources that both the processor and the basic block itself need to run properly, as well as a control logic to ensure that the functionalities of the basic block are preserved (or not) throughout the course of the fault injection campaign. As Shown in fig:2 a **random** initialization is included in the header for the basic block, ensuring the non dependability of the reliability metrics extracted on the input data, together with a footer that checks the functionalities of the block itself. Notice that in in this case, contrary of what is done in the Hardware methodology, there is no physical probing of the circuit on which the program or the testbench is running. In this study only the functional aspect of the Software Product under test is observed.

B. Fault Injection on Randomly Initialized Resources

Fault injection is the mean by which the misbehavior and faulty execution is provoked on purpose on digital systems. In the past, especially on hardware, fault injection was aimed to functionally verify the designs under test. Those DUT were analysed, their functions (data dependent) extracted and inputs were selected in order to exercise those functions. Later on the fault injection had the role of determining whether those functions were preserved in cases of fault or how eventually they were modified. Today this is still the state of the art for software verification.

With time a second approach on hardware was presented, testing moved from functional to structural, where the integrity of the device is evaluated, regardless of the function (and

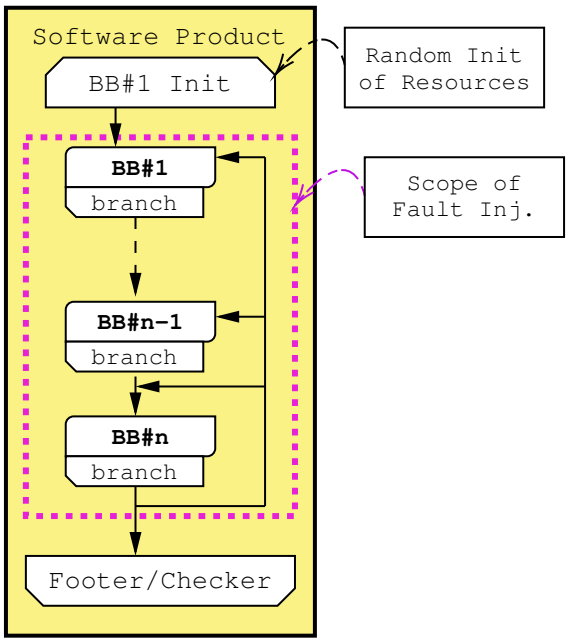

Fig. 2: Block Diagram of the Entire SW Product

therefore of data), verifying solely the implemented boolean function.

The methodology introduced in this paper presents the novelty of applying this structural approach to software. To abstract the basic block as much as possible from its link to data, **every resource utilized has been randomized** before each fault injection. The probability of failure and propagation probabilities are therefore extracted **independently** of their data input.

Probes (i.e., observation points) are defined during the setup of the fault injection campaign. It is the role of the Footer/Checker (out of the scope of the fault injection) to redirect the output of the block function into a reserved portion of the memory to be probed. Probes are set on those reserved memory location on all blocks, not probing the correctness of the data with respect to the golden run, but solely if the basic function included in that portion of code has been affected by the fault injection.

C. Re-Composition of the basic blocks

Once the fault injection campaigns are over, it is time to re-compose the information that have been extracted on the single blocks into a complete description of the original Software Product. To perform the re-composition there is the need run the software once and record the trace, this will allow us to know exactly the sequence in which the basic blocks have been executed during the nominal run.

We distinguish two main branches of the re-composition, the ones containing fault that *do not modity the program flow* and those that lead to a *modified program flow*

1) Not modified Program Flow: First we nee to define the probability of being executing a precise basic block in time during the execution of the program. Assuming a deterministic duration per executed instruction, without nested or hidden operation, we can define the probability of executing BB_n as

$$P_{in-bb_n} = \frac{instructions - in - BB_n}{total - instruction - in - exe} \quad (4)$$

Next step is to define the probability of a fault happening in BB_n being able to become an error in the same block. This has

been deduced from fault injection and must be differentiated per every register in which we inject faults and it represented as:

$$P^{A_m}_{gBB_n} \tag{5}$$

Last probability to define is the probability of a block to receive a wrong input and propagate it to its output. Defined as:

$$P^{A_m}_{pBB_n} \tag{6}$$

which is related to the "time of life" of the variables, defined as the number of basic block between the last time a variable has been read and the first time it gets overwritten.

Once these probabilities ha been defined we can describe the worst possible case, in which a fault is injected in BB_n and gets propagated throughout the whole program.

$$\begin{aligned} &P_{in-bb_n} * P^{A_m}_{gBB_n} * \left[P^{A_m}_{pBB_{n+1}} \dots P^{A_m}_{pBB_f}\right] + \\ &+ P_{in-bb_{n+1}} * P^{A_m}_{gBB_{n+1}} * \left[P^{A_m}_{pBB_{n+2}} \dots P^{A_m}_{pBB_f}\right] \dots \end{aligned} \tag{7}$$

which summarizes, per every register A_m as:

$$\sum_{n=0}^{N} P_{in-bb_n} * P^{A_m}_{gBB_n} * \left[\prod_{i=n}^{N} \left[P^{A_m}_{pBB_{i+1}}\right]\right] \tag{8}$$

$$P_{tot} = [P_{in-bb-x} * P_{p-bbx}] + [P_{p-bbx+1} * P_{p-bbx+1}] \dots \tag{9}$$

2) Modified Program Flow: Regarding the possibility of having a fault injected on a register while the program is executing a precise Basic Block that requires a branching operation at the end, we cannot consider them while recomposing the metrics as in the previous subsection.

These blocks contribute instead to the composition of a particular subset of runs (diverse behaviour of the program) which include all those runs in which the program simulation has reached the end in a time that differs from the nominal one. In particular, it can be shortened due to a premature jump to the conclusive part of the program as well as delayed due to an incorrect loop that sends the machine into a non-necessary series of states from which it will eventually recover. In the case in which the machine would not be able to recover, we categorize those runs as Timeouts (when longer than 150% of nominal time). *It worth to point out that, due to the nature of the injections, which focus on the Register file, with one SEU per run, these cases are reduced to the minimum, if not nonexistent.* Give these assumptions, taking into account this second section of Basic Blocks, it is possible to assume that most, if not all of these runs will generate a failure in the functionalities of the program itself. therefore the recomposition, that was missing a good half of what was neded, now finds the missing cases in all those blocks that led to a modification in the flow.

In particular, considering the possibility that this blocks have not to propagate (to mask) a fault occurring in the course of their routine, the event of flow corruption has probability $1 - P_{masking}$, then the probability of these fault becoming a funcional error is 100% and it does not propagate. In this case the recomposition technique is slightly different than the previous section, as the case of a missed branch or jump leads directly to an error. So defined the multiplicity of the same critical block in the nominal sequence m, the probability of having a functinoal failure is described by:

$$P_{err} + P_{msk} * P_{err} + (P_{msk})^2 * P_{err} \dots + (P_{msk})^m * P_{err} \tag{10}$$

Taking into accoun the approximation due to the algorithm intrinsic ability to recover from a flow error.

IV. Test Case and Application

A. The Software

The Software of choice for the Proof of Concept of this methodology is the Bubble Sort Algoritm, in its Assembler for RISC-V Version of the C code reported Below.

```c
#include <stdio.h>

int main()
{
    int arr[5];
    arr[0] = 3;
    arr[1] = 5;
    arr[2] = 1;
    arr[3] = 2;
    arr[4] = 4;

    int tmp;

    for(int i = 0; i < 4; i++){
        for(int j = 0; j < 4 - i; j++){
            if(arr[j] > arr[j + 1]){
                tmp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = tmp;
            }
        }
    }
    return 0;
}
```

Listing 2: Implemented Algoritm

Bubble sort is an $O(n^2)$ sorting algorithm. A simple sorting algorithm that performs a one-way comparison of two adjacent records from the head to tail of the disordered part in each sort trip. Of course, the direction can also be the contrary, one-way comparison from the tail to head of the disordered part. This will form gradually an ordered table at the head of the disordered table, and the basic idea of the algorithm has no difference with the foregoing [18]. The implementation in Assembler is not reported in this paper due to lenght, but the code results to be organized as follow:

- **.main** - Create stack and initialize the parameters arr[] and i in stack memory
- **.L2** - If i $<$ 4 jamp to .L6, else free the stack memory then return
- **.L6** - Initailize parameter j in stack memory
- **.L3** - If (j - i) $<$ 4 jump to .L5, else i++ and go to .L2 to check i
- **.L5** -
  - Compare arr[j] to arr[j+1]
  - If arr[j+1] $<$ arr[j]
    - Store arr[j] to tmp
    - Assign arr[j+1] to arr[j]
    - Assign tmp to arr[j+1]
    - Go to .L4 to execute j++ then check j
  - If arr[j+1] $>$ arr[j]
    - Jump to .L4 and execute j++ then check j
- **.L4** - Jump to .L4 and execute j++ then check j

B. The Division in Basic Block

The processing of dividing the Software under test into basic block has been carried out automatically and returned 8 different blocks, together with the list of resources that each and every basic blocks utilizes during its own functions. After a Nominal run without faults of the entire software, it was possible to trace the transition between the different basic blocks throughout the whole execution. These information, summarized in the scheme below, will be the key to predict the behaviour of the program starting from the behaviour of the basic blocks themselves.

TABLE I: Result of Fault Injection on basic blocks

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	x10	x11	x12	x13	x14	x15	x16		x31
bb_0	0	0	139	0	0	0	0	0	379	0	0	0	0	0	0	162	0	...	0
bb_1	0	0	0	0	0	0	0	0	500	0	0	0	0	0	0	0	0	...	0
bb_2	0	0	0	0	0	0	0	0	269	0	0	0	0	28	14	38	0	...	0
bb_3	0	0	0	0	0	0	0	0	500	0	0	0	0	16	131	244	0	...	0
bb_4	0	0	0	0	0	0	0	0	495	0	0	0	0	0	0	62	0	...	0
bb_5	0	0	0	0	0	0	0	0	380	0	0	0	0	0	0	0	0	...	0
bb_6	0	0	0	0	0	0	0	0	494	0	0	0	0	0	0	41	0	...	0
bb_7	0	0	0	0	0	0	0	0	466	0	0	0	0	0	0	0	0	...	0

TABLE II: Comparison of Fault Injection data vs Recomposed data on Entire Software

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	x10	x11	x12	x13	x14	x15	x16		x31
FI	0	0	4	0	0	0	0	0	221	0	0	0	0	15	41	90	0	...	0
Reco	0	0	5	0	0	0	0	0	228	0	0	0	0	13	44	98	0	...	0

TABLE III: Control Flow Driven Errors

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	x10	x11	x12	x13	x14	x15	x16		x31
FI	0	0	0	0	0	0	0	0	198	0	0	0	0	4	77	90	0	...	0

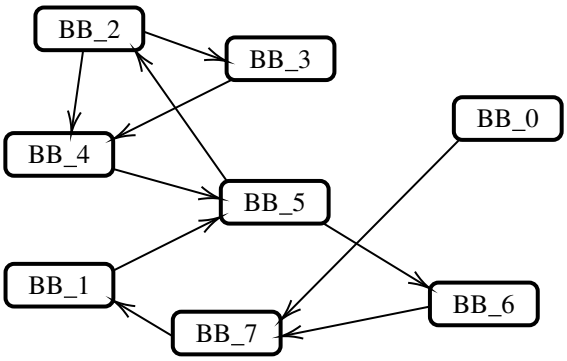

Fig. 3: Block Diagram of the Entire SW Product

C. The Platform

The choice of the platform on which the program has been run and tested fell on the SCR1SOC. SCR1 is an open-source and free to use RISC-V compatible MCU-class core, designed and maintained by Syntacore. It is industry-grade and silicon-proven (including full-wafer production), works out of the box in all major EDA flows and Verilator, and comes with extensive collateral and documentation []. This choice had mostly been driven by the larger and larger usage of these kind of RISCV based cores in the academic community. Any test based on these platforms is and added value to their development.

D. The Fault Injection Campaign

The next step in the methodology is fault injection. It is performed using Cadence fault injection tool *FSV* [15]. Once fault injection sites are automatically identified from the RTL description, fault injection is performed and 20 faults are injected per identified site using a custom pre-generated fault dictionary, including a random injection time. An in-house tool build on top of the *GSL* [19] has been developed for this purpose. Such number is statistically significant enough [20] without compromising fault injection campaign running time. Faults are injected on the integrity of the register file, mimic as well the possibility of faults propagated to the memory and back. Also, fault probes are set on non excerciced memory location to record which injected faults will cause a functional failure of the basic block. For each fault injection run, a logfile is generated which reports the outcome of the run, later a custom made parsing tool will recollect the data from this logfile and present the results to the re-composition tool.

V. Results and Future Work

The results of the recomposition are based on Table:1, which summarizes the result of the fault injection campaign that has been carried out on the single basic blocks. Each entry of the table enumerates the number of functional error on caused by each register in the register file, keeping in mind that every bit in the register has been affected by 20 faults randomized in time, for a total of 640 faults per register. Once these table has been given to the recomposition tool, Table:2 is returned, including the benchmark fault injection campaign on the entire Software Product for validation of results together with the expected number of faults, calculated following the methodology described. Last, Table:3 Reports the number of Errors that have been caused by an error in the flow of the program, which can be extracted by an equivalent of table number 1 for flow errors caused by each register failing in each basic block and recomposed as in its dedicated section.

The last part of the methodology will be the focus of the work to come, as includes the implicit ability of the different algorithms to recover from flow errors, which understanding can lead to much more refined results.

References

[1] Eric Cheng, Shahrzad Mirkhani, Lukasz G. Szafaryn, Chen-Yong Cher, Hyungmin Cho, Kevin Skadron, Mircea R. Stan, Klas Lilja, Jacob A. Abraham, Pradip Bose, and Subhasish Mitra. Clear: Cross-layer exploration for architecting resilience: Combining hardware and software techniques to tolerate soft errors in processor cores. In 2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC), pages 1–6, 2016.

[2] Jörg Henkel, Lars Bauer, Nikil Dutt, Puneet Gupta, Sani Nassif, Muhammad Shafique, Mehdi Tahoori, and Norbert Wehn. Reliable on-chip systems in the nano-era: Lessons learnt and future trends. In 2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC), pages 1–10, 2013.

[3] Jinhee Park, Hyeon-Jeong Kim, Ju-Hwan Shin, and Jongmoon Baik. An embedded software reliability model with consideration of hardware related software failures. In 2012 IEEE Sixth International Conference on Software Security and Reliability, pages 207–214, 2012.

[4] Mojtaba Ebrahimi, Abbas Mohammadi, Alireza Ejlali, and Seyed Ghassem Miremadi. A fast, flexible, and easy-to-develop fpga-based fault injection technique. Microelectronics Reliability, 54(5):1000–1008, 2014.

[5] Mojtaba Ebrahimi, Maryam Rashvand, Firas Kaddachi, Mehdi B. Tahoori, and Giorgio Di Natale. Revisiting software-based soft error mitigation techniques via accurate error generation and propagation models. In 2016 IEEE 22nd International Symposium on On-Line Testing and Robust System Design (IOLTS), pages 66–71, 2016.

[6] Maha Kooli, Firas Kaddachi, Giorgio Di Natale, Alberto Bosio, Pascal Benoit, and Lionel Torres. Computing reliability: On the differences between software testing and software fault injection techniques. Microprocessors and Microsystems: Embedded Hardware Design (MICPRO), 50:102–112, May 2017.

[7] M. Blackburn, R. Busser, A. Nauman, R. Knickerbocker, and R. Kasuda. Mars polar lander fault identification using model-based testing. In Eighth IEEE International Conference on Engineering of Complex Computer Systems, 2002. Proceedings., pages 163–169, 2002.

[8] Tiziano Fiorucci, Jean-Marc Daveau, Giorgio di Natale, and Philippe Roche. Automated dysfunctional model extraction for model based safety assessment of digital systems. In 2021 IEEE 27th International Symposium on On-Line Testing and Robust System Design (IOLTS), pages 1–6, 2021.

[9] Maha Kooli and Giorgio Di Natale. A survey on simulation-based fault injection tools for complex systems. In 2014 9th IEEE International Conference on Design Technology of Integrated Systems in Nanoscale Era (DTIS), pages 1–6, 2014.

[10] A. Vallero, S. Tselonis, N. Foutris, M. Kaliorakis, M. Kooli, A. Savino, G. Politano, A. Bosio, G. Di Natale, D. Gizopoulos, and S. Di Carlo. Cross-layer reliability evaluation, moving from the hardware architecture to the system level: A {CLERECO} {EU} project overview. Microprocessors and Microsystems, 39(8):1204 – 1214, 2015.

[11] A. Vallero, A. Savino, A. Chatzidimitriou, M. Kaliorakis, M. Kooli, M. Riera, M. Anglada, G. Di Natale, A. Bosio, R. Canal, A. Gonzalez, D. Gizopoulos, R. Mariani, and S. Di Carlo. Syra: Early system reliability analysis for cross-layer soft errors resilience in memory arrays of microprocessor systems. IEEE Transactions on Computers, 68(5):765–783, 2019.

[12] Shubu Mukherjee. Architecture Design for Soft Errors. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[13] N.J. Wang and S.J. Patel. Restore: Symptom-based soft error detection in microprocessors. IEEE Transactions on Dependable and Secure Computing, 3(3):188–201, 2006.

[14] S. Mirkhani, M. Lavasani, and Z. Navabi. Hierarchical fault simulation using behavioral and gate level hardware models. In Proceedings of the 11th Asian Test Symposium, 2002. (ATS '02)., pages 374–379, 2002.

[15] Cadence. https://www.cadence.com/en_US/home.html.

[16] Synopsys zo1x. https://www.synopsys.com/verification/simulation/z01x-functional-safety.html.

[17] Marco Ottavi, Dario Asciolla, Tiziano Fiorucci, Elena Grosso, Carla Marzullo, Alessandro Scaramella, Simone Stramaccioni, Alessia Zibecchi, Carla Andreani, Gian Carlo Cardarilli, Carlo Cazzaniga, Luca Di Nunzio, Rocco Fazzolari, Marco Re, Pedro Reviriego, Gianluca Furano, and Roberto Senesi. Setup and experimental results analysis of cots camera and srams at the isis neutron facility. In 2018 13th International Conference on Design Technology of Integrated Systems In Nanoscale Era (DTIS), pages 1–4, 2018.

[18] Wang Min. Analysis on bubble sort algorithm optimization. In 2010 International Forum on Information Technology and Applications, volume 1, pages 208–211, 2010.

[19] Gsl, gnu scientific library. https://www.gnu.org/software/gsl/.

[20] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert. Statistical fault injection: Quantified error and confidence. In 2009 Design, Automation Test in Europe Conference Exhibition, pages 502–506, 2009.