BAFFI: a bit-accurate fault injector for improved dependability assessment of FPGA prototypes

Ilya Tuzov[1], David de Andrés[2], Juan-Carlos Ruiz[2], and Carles Hernández[1]
[1]DISCA, Universitat Politècnica de València, Campus de Vera s/n, 46022, Spain
[2]ITACA, Universitat Politècnica de València, Campus de Vera s/n, 46022, Spain
tuil@upv.es, ddandres@disca.upv.es, jcruizg@disca.upv.es, carherlu@upv.es

***Abstract*—FPGA-based fault injection (FFI) is an indispensable technique for verification and dependability assessment of FPGA designs and prototypes. Existing FFI tools make use of Xilinx essential bits technology to locate the relevant fault targets in FPGA configuration memory (CM). Most FFI tools treat essential bits as black-box, while few of them are able to filter essential bits on the area basis in order to selectively target design components contained within the predefined Pblocks. This approach, however, remains insufficiently precise since the granularity of Pblocks in practice does not reach the smallest design components. This paper proposes an open-source FFI tool that enables much more fine-grained FFI experiments for Xilinx 7-series and Ultrascale+ FPGAs. By mapping the essential bits with the hierarchical netlist, it allows to precisely target any component in the design tree, up to an individual LUT or register, without the need for defining Pblocks (floorplanning). With minimal experimental effort it estimates the contribution of each DUT component into the resulting dependability features, and discovers weak points of the DUT. Through case studies we show how the proposed tool can be applied to different kinds of DUTs: from small-footprint microcontrollers, up to multicore RISC-V SoC. The correctness of FFI results is validated by means of RT-level and gate-level simulation-based fault injection.**

***Keywords*—Fault injection, FPGA, configuration memory, robustness assessment, RISC-V**

I. Introduction

Current FPGA fault injection (FFI) methodologies emulate logic faults in FPGA prototypes by manipulating the content of their configuration memory (CM) at runtime. In that context, many FFI tools targeting older FPGA series were capable of emulating logic faults at the granularity of individual netlist cells[1]. To this end, they relied on the vendor-specific frameworks like Jbits to automate netlist modifications, and reflect them directly into the differential/partial bitstreams [2]. Despite modern FPGA frameworks, like Xilinx Rapid Wright [3], allow similar netlist modifications, they are unable to reflect these changes into the bitstream.

At the same time, the bitstream format of modern FPGAs is documented rather skimpily. This prevents designers from manually deducing which bitstream modifications are required for targeted emulation of logic faults at the netlist level. Despite some information on bitstream format is available, e.g. from the X-Ray Project or related research works [4][5], it still remains scarce and mostly targets outdated FPGA series (Xilinx 7-series and older). To the best of author's knowledge, there is no publicly available FFI tool for modern FPGAs, that would be capable of fine-grained fault injections at the level of hierarchical netlist.

For that reason, most modern FFI tools focus on emulation of random bit-flips at the level of the CM itself, without establishing any relation between the CM and design under study (DUT), other than area/Pblock-based mapping of essential bits. Essential bits [6] are a subset of CM cells that determine the circuitry of the DUT in FPGA. Some recent FFI tools [7][8] treat essential bits as black-box, so to evaluate the effect of CM faults on the DUT, but without providing any insights on the dependability features of individual DUT components. Few FFI tools [9][10] are able to filter essential bits attending to the rectangular chip area (Pblock), thus reaching finer granularity of fault injection (grey-box approach). However, one major disadvantage of these tools is that their precision is conditioned by the granularity of defined Pblocks, remaining in practice rather coarse-grained, at least not reaching individual netlist cells. In addition, they present high level of intrusiveness, since any Pblock alters the placement-routing results of FPGA design with respect to unconstrained design.

In this paper, we propose BAFFI, a bit-accurate FFI tool that supports fine-grained fault injection at the level of hierarchical netlist. The underlying FFI methodology is based on a bit-accurate mapping of essential bits with a hierarchical netlist, covering main types of FPGA cells, namely LUTs, registers, LUTRAM and BRAM.

The main advantage of BAFFI is its ability to target any component in the DUT hierarchy (up to an individual LUT or register) without the need for floorplanning, which is significantly important when assessing the effectiveness of specific fault-tolerant mechanism implemented in the design, or to obtain detailed robustness estimates for each component in the DUT tree. In addition, BAFFI is highly customizable, allowing to build FFI setups controlled from the host PC or from the FPGA itself, supporting diverse Xilinx FPGAs (including 7-series and Ultrascale+) and DUTs of any complexity.

II. Emulating upsets in changeable and non-changeable CM cells

A typical FPGA bitstream contains two types of configuration data: the changeable and non-changeable CM bits. The changeable memory bits contain the initial content of registers, BRAMs, and distributed RAMs (LUTRAMs). Their content dynamically changes during circuit operation, and their current

978-3-9819263-7-8/DATE23/© 2023 EDAA

state can be examined by means of readback-capture procedure [11]. Likewise, their content can be modified (manipulated for FFI purposes) by means of readback-modify-write procedure. It is worth noting that any data manipulation on changeable memory requires prior pausing of clock signal in order to prevent undesired data corruption. The location of changeable memory bits in the CM (bitstream) can be extracted from the logic location (.LL) file, which is exported by the Vivado suite alongside the bitstream for Xilinx devices. For each register, BRAM and LUTRAM this file reports a CM frame address (FAR) and an offset within the frame that corresponds to each particular bit of these memory elements.

Injection of bit-flip faults in each of these elements requires several steps. For instance, flipping the register state requires to: (i) pause DUT clocking, (ii) activate *GCAPTURE* signal to save the current register state into the associated (INIT) CM cell, (iii) readback the CM frame that is listed in the LL file for the targeted register, (iv) modify (invert) the bit at the corresponding offset (also listed in LL file), (v) write back the modified data frame, (vi) activate *GRESTORE* signal to set the state of the targeted register from the updated INIT cell.

The non-changeable memory data constitutes the rest (major part) of the bitstream. It configures the combinational logic (LUTs, MUXes, carry chains), as well as wiring/routing resources (including switchboxes). Emulation of upsets in non-changeable CM is simpler since it does not require capture/restore steps. However, the content of this memory is not documented by FPGA vendors (e.g Xilinx) in any way other than format of CM address space [12]. Though, some scarce information can be found from the related research, e.g. on the location of LUT configuration bits for Xilinx 6-series and 7-series FPGAs [4][13]. The only useful aid provided by Vivado [1] for the non-changeable memory is the essential bit mask file (*.ebd), which highlights those CM cells that determine the functionality and integrity of the circuit in FPGA. This file is originally intended for usage with the Xilinx Single Error Mitigation (SEM) IP [14] both for scrubbing and fault injection purposes. Some works in the field make use of this file in their custom FFI tools, albeit mostly based on SEM IP as well. The problem is that being unable to relate the essential bits with the DUT hierarchy, these FFI tools either blindly target all essential bits in a statistical way [7], or at best filter them on the area basis [9], but still not relating them to DUT hierarchy. Unfortunately, statistical FFI is useful for robustness estimation of designs targeting FPGAs but not to test the dependability and behaviour of the different safety elements in a SoC or to trigger potential fault-induced security vulnerabilities that can occur when targeting specific bits [15].

III. Bit-accurate FFI tool

BAFFI tool offers fully automated experimental flow as depicted in Fig.1. It comprises four phases: (i) mapping of essential bits with a hierarchical netlist, (ii) generation of faultlist (sampling of essential bits) attending to the configured filters

[1]Vivado is the Xilinx FPGA synthesis tool.

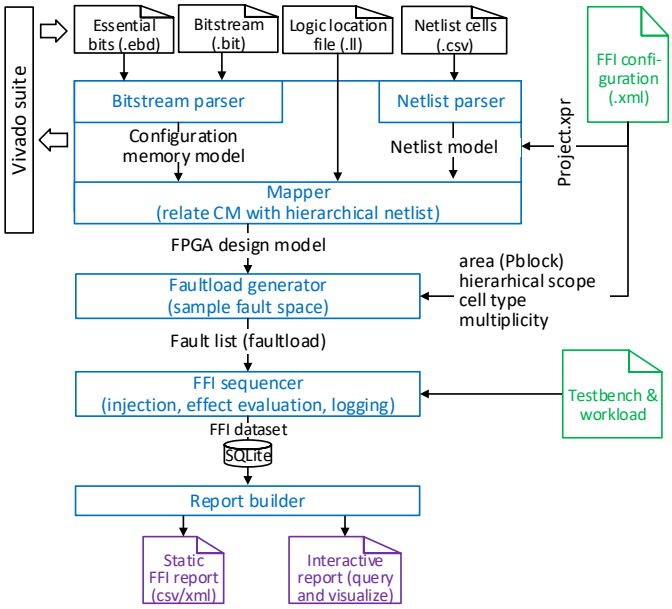

Fig. 1. Workflow and main components of the BAFFI tool

and parameters, (iii) execution of FFI runs, including injection of sampled faults and evaluation of fault effects on the DUT behaviour, (iv) generation of FFI report and visualization of logged FFI results through the interactive reporting interface. The user is only in charge of (i) configuring the faultload parameters in an XML-formatted file, (ii) adapting a testbench template for the targeted DUT, and (iii) invoking BAFFI from the command line terminal with the XML configuration file on the input. This section details each of the aforementioned FFI phases.

A. Mapping of essential bits

The objective of the mapping phase is to relate the hierarchical path of netlist cells with the addresses of corresponding essential bits, as it is depicted in Fig.2. This process is managed by the *Design Parser* module. Process starts by parsing the debug bitstream (in which each data frame is annotated by a frame address) to extract the list of valid frame addresses for the target FPGA part. This list of frame addresses is subsequently used to parse the essential bits (EBD) mask and to relate it with the bitstream data under the configuration memory model. The post-place-route design is processed by a custom Vivado script to extract a set of logic and placement attributes into an internal netlist model. Among others, these attributes include: a hierarchical path in the design tree, X:Y coordinates of a Tile and a Slice on the FPGA floorplan, BEL Label within the Slice (e.g. A6LUT, AFF), and the INIT value.

Once the CM and netlist models are generated, they are mapped together by applying a set of FPGA-specific rules. Some of these rules have been derived by quick coarse-grained bitstream analysis. For instance, in the 7-series devices one *Major Frame* of CM covers two columns of CLB slices and one column of interconnecting switchboxes (SW), and the *Major Frame* index equals the X_{TILE} coordinate of a CLB

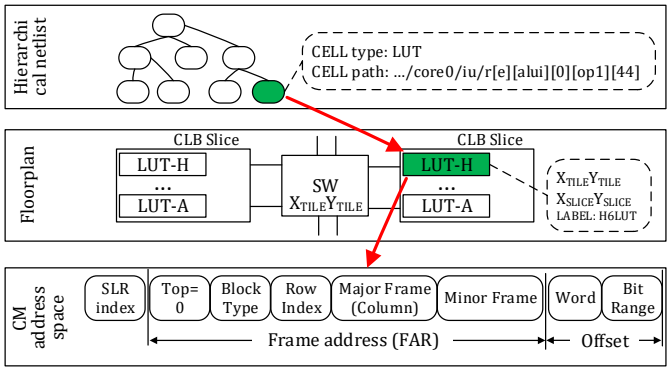

Fig. 2. Mapping between netlist cells and essential bits: from the hierarchical name to the CM address

Slice. In the Ultrascale+ devices, CLB and SW columns are configured by different *Major Frames*. In this case, the mapper first looks up in the bitstream for the indices of those *Major Frames* that configure the switchboxes $M_i = SW(X_{TILE})$. These major frames are easily distinguished by their size (they contain 76 minor frames). Subsequently, the *Major Frame* indices of connected CLB (BRAM, DSP) slices are calculated as $M_i - 1$ for the left slice, and $M_i + 1$ for the right slice.

It is worth noting, that the FAR fields *Top* and *Row index* are calculated from the Y coordinate of the clock region (*Top* field is not applicable to Ultrascale FPGAs), and the *Block type* field equals 0 for all CM bits except BRAM content (whose *Block type* equals 1).

We have derived the rules of fine-grain mapping experimentally by low-level bitstream analysis, and currently cover those netlist elements that form a major part of sequential and combinational logic in FPGAs (registers, LUTs, BRAMs and LUTRAMs). In particular, the essential bits of LUTs can be localized within the major frames by translating the LUT attributes Y_{TILE} and $LABEL$ into a list of tuples *[Minor Frame, Word, Bit]*, in total 64 tuples for LUT6 cell. This mapping is depicted in Fig.3 for 7-series devices, and in Fig.4 for Ultrascale+. The mapping we have derived for 7-series devices matches the one found by [13][4]. However, up to our knowledge, we provide the first mapping for the Ultrascale+. It can be seen that in both FPGA series the content of LUTs spans across four minor frames, although the frame indices and word offsets are different.

Despite the mapping of LUTRAMs, BRAMs, and registers can be calculated from their placement coordinates (similarly to LUTs), the BAFFI tool extracts this mapping directly from the .LL file (that contains location of DUT's changeable memory bits). This file has human-readable ASCII format, and its parsing is quite straightforward using regular expressions.

As a result of the mapping step, an FPGA design model is created that indicates the location of essential bits of each LUT, Register, BRAM and LUTRAM in the design tree. The rest of FPGA resources that are not currently covered by bit-accurate mapping, are related with their essential bits at the Pblock-granularity (similarly to the state-of-the-art FFI tools).

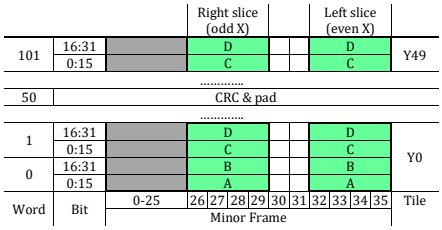

Fig. 3. Location of LUT content in the CM of 7-series FPGA (4 LUTs per CLB slice labeled A–D)

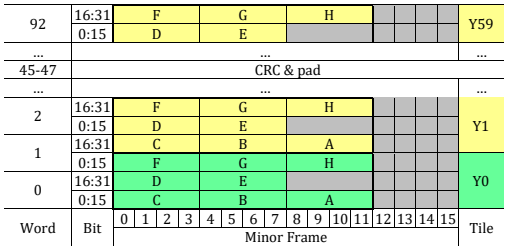

Fig. 4. Location of LUT content in the CM of Ultrascale+ FPGA (8 LUTs per CLB slice labeled A–H)

B. Faultload generation

The faultload generator relies on the obtained design model to a create a fault list attending to filters and parameters, configured in the input XML file. Such filters include:

- *design_scope* specifies the list of targeted DUT nodes, defined by their hierarchical path in the DUT tree;
- *pblock* specifies a rectangular FPGA area, defined by the XY coordinates of bottom-left and top-right Tiles, or by the name of Pblock used in the Vivado design;
- *target_logic* specifies the type of targeted FPGA resources, including (i) *LUT*, *Register*, *LUTRAM*, *BRAM* for bit-accurate FFI, or (ii) *Type-0* for the entire set of essential bits including logic and routing selected with Pblock granularity.

The *mode* attribute configures the way fault configurations are selected from the fault space: (1) statistical sampling of *sample_size* faults attending to the approach in [16], (2) iterative sampling until reaching the predefined *error_margin* attending to the approach in [17], (3) exhaustive mode to test all essential bits in the selected DUT scope. A *fault_multiplicity* attribute configures the number of faults emulated during each individual injection run. Finally, a *CCF* attribute specifies a list of structurally identical DUT components that can be targeted for the emulation of common-cause faults [18].

The generated fault list is exported to the binary .dat file and uploaded to the FPGA-side FFI controller. In the case of board-controlled injection (see section III-C), this fault list can be generated directly by FFI controller on the basis of essential bits mapping. Each fault configuration in a list includes a CM address of targeted essential bit and a fault injection time (measured at clock cycle granularity), or a set of such items in case of multi-bit faults.

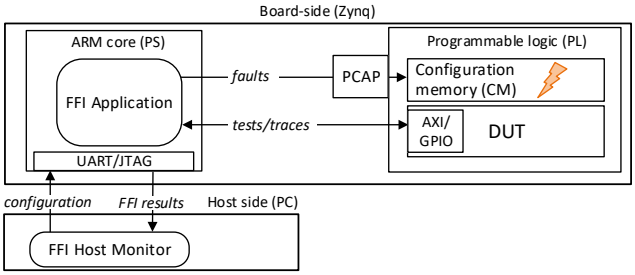

Fig. 5. FFI setup controlled by board-side FFI application

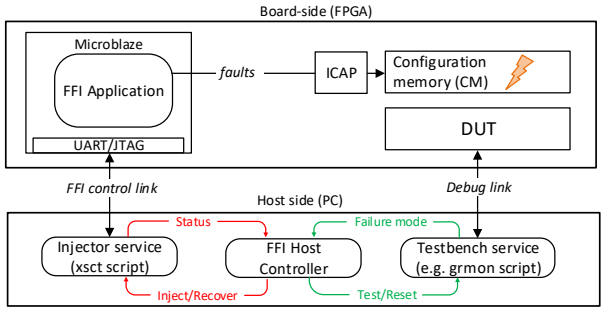

Fig. 6. FFI setup controlled by host-side FFI application

C. Fault injection and effect evaluation

The generated faultlist is processed by the FFI sequencer during fault injection runs. Each injection run comprises five steps: (i) initialize the DUT, (ii) run the workload on the DUT until the fault injection time, (iii) modify CM content attending to the fault configuration (fault injection), (iv) trace DUT behaviour and determine the effect of injected fault (failure mode), (v) remove the fault (recover CM content) and reset the DUT. BAFFI allows to set up this process in two ways: under the control of board-side FFI application (Fig.5), or under the control of host-side FFI application (Fig.6).

The board-controlled FFI setup is appropriate for those DUTs that can be tested without host intervention. In this case, all FFI steps are autonomously executed on the board side, using a hardwired ARM core (in Zynq FPGAs) or a Microblaze IP (in Virtex/Kintex FPGAs) as an FFI controller. The task of host application is limited to initializing the embedded Zynq/Microblaze application, supplying it with input data (fault list), and monitoring (collecting) FFI results logged by the onboard controller.

The host-controlled setup is used for those DUTs that require a debug link with the host PC. Such as, for instance, a Cobham Gaisler's LEON5 and NOELV processors that are initialized and tested by means of GRMON tool. In this case the board-side FFI controller is in charge of only fault injection (performing CM manipulations and controlling DUT clocking) on the request of the host application. Whereas the evaluation of fault effects is performed by a testbench service on the host-side. On the request of main FFI application, the testbench invokes an executable (workload) on the DUT, evaluates the DUT outputs and (optionally) its internal state, determines the failure mode and returns it to the main application as an outcome of an FFI run.

The board-controlled setup reduces to the minimum the fault injection and effect evaluation latencies. This enables higher experimental speed than in the case of host-controlled FFI. The latter, on the other hand, is more flexible, allowing integration of standard debuggers into the FFI flow.

IV. Experimental Evaluation

BAFFI supports a wide range of experimentation scenarios. This section exemplifies two particular use cases: (i) fault sensitivity analysis of a multicore RISC-V CPU by means of fine-grained FFI, and (ii) dependability benchmarking of soft-core microcontrollers. The correctness of BAFFI results is validated by means of simulation-based fault injection (SFI). Finally, this section discusses the experimental speed achievable by host-controlled and board-controlled FFI setups, and compares it with the experimental speed of RTL and gate-level SFI.

A. Characterizing fault sensitivity of NOELV processor core

NOEL-V is a RISC-V processor developed by Cobham Gaisler [19]. The processor configuration selected for this case study implements a 64-bit RISC-V core [20] with dual-issue pipeline, integer, floating point, atomics and multiply and divide extensions (a.k.a IMAFD). A four-core configuration of this processor has been implemented on the Virtex Ultrascale+ FPGA (Xilinx VCU118 evaluation board). Each core is placed on its own Pblock during the implementation in Vivado (version 2021). Each core of this processor runs an integer matrix multiplication workload.

NOEL-V testing environment requires a host-side debug monitor GRMON [21]. For that reason, a host-controlled BAFFI setup is used in this case study (Fig.6). In the testbench template (represented by a GRMON script) for this DUT, we customize two functions that are responsible for (i) resetting the DUT, and (ii) evaluating the processing results and detecting abnormal DUT states (e.g. crashes/hangs). Both reset and test functions are invoked on the request of main BAFFI application, using Linux sockets for communication. The processing results are stored at the predefined memory area; the testbench accesses and verifies these results during each FFI run, and determines the failure mode: (i) *masked* when the processing results are correct and no abnormal DUT state has been detected, (ii) *silent data corruption (SDC)* when the processing results are incorrect but the DUT operation continues without alerts, and (iii) *crash* in the case of abnormal DUT state preventing it from further operation (requiring hardware reset), e.g. absence of DUT response.

Faults are injected into core-0 of NOEL-V CPU, attending to a common grey-box approach and to a proposed bit-accurate approach. In the former case, FFI targets all essential bits in the area of core-0, i.e. by applying filter *pblock="core:tiles:X2Y302:X90Y359"*, BAFFI reports roughly 14.9 Mbits of essential bits. In the latter case, FFI targets core-0 by a hierarchical path, i.e. by applying filter

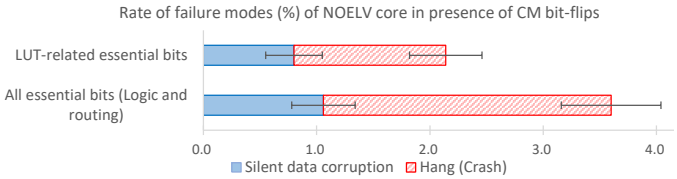

Fig. 7. Robustness estimates of NOELV core obtained after the common area-based FFI approach

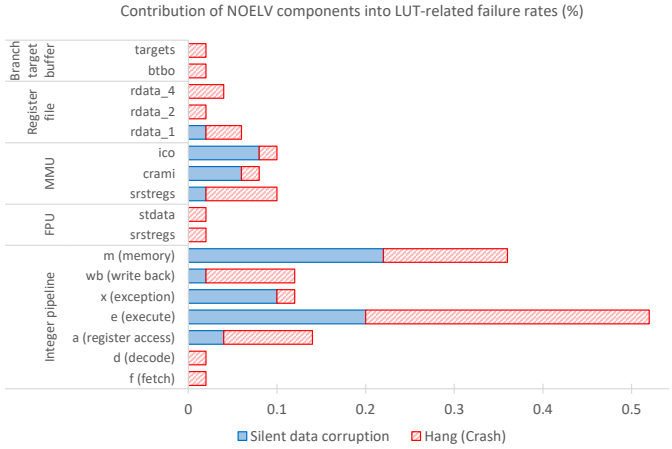

Fig. 8. Detailed robustness estimates of NOELV core obtained after the bit-accurate FFI approach

dut_scope="cpu/core0/gpp0/noelv0/cpuloop[0].core", BAFFI reports roughly 3.1 Mbits of essential bits (48279 LUTs). Both experiments perform statistical sampling of 5000 faults.

Dependability estimates obtained after the common area-based approach (depicted in Fig.7) show an SDC rate of 1.06% and crash rate of 2.54% for the entire set of essential bits (including logic and routing) in CPU core-0. When area-based approach targets only LUT-related essential bits, the SDC rate is reduced to 0.8% and crash rate is reduced to 1.34%, showing that upsets in LUTs are less critical than upsets in the routing-related CM.

Estimates provided by the bit-accurate FFI are depicted in Fig.8. These detailed results reveal how much each component of NOEL-V core contributes into the previously described Pblock estimates. For instance, it can be seen, that the integer pipeline is the most critical core component, contributing more than a half of SDCs (72%) and crashes (53%). Among the pipeline stages, the most critical ones are the memory access (m) and the execute (e). The second most critical core component is the MMU that contributes roughly 15% of SDCs and 9% of crashes to the total LUT-related core-0 estimates. These fault sensitivity results can be further traced along the DUT tree, up to a granularity of each individual LUT bit. More importantly, this experiment shows that BAFFI enables testing of individual components in large SoCs. This is especially important in the context of safety-related applications in which safety mechanisms need to be placed to avoid safety goal violations.

B. Dependability benchmarking of soft-core microcontrollers

The second case study analyzes the dependability features of three small-footprint soft-core processors: MC8051 [22], AVR [23], and Microblaze [24]. The complexity of these processors is low enough as to make feasible simulation-based fault injection (SFI) experiments at RTL and gate-level. The objective of the case study is to validate the dependability estimates reported by BAFFI by cross-comparing them with the estimates obtained by SFI (using an SFI tool in [25]).

All three DUTs run a matrix multiplication workload. The faultload includes (a) bit-flips in registers injected at RTL, gate-level and FPGA-level, with an error margin on 0.5%, (ii) bit-flips in BRAM and LUTRAM injected at gate-level and FPGA with an error margin of 0.1%, and (iii) upsets in LUTs at gate-level (only in case of MC8051) and FPGA-level sampled with an error margin of 0.1%.

None of considered DUTs requires a debug link with the host. Thus, for the sake of higher experimental speed a board-controlled BAFFI setup is used in this experiment. A 7-series Zynq FPGA is used in this experiment (Xilinx ZC702 evaluation board), whose hardwired ARM core operates as FFI controller for both fault injection and effect evaluation (testing) purposes as depicted in Fig.5.

TABLE I. SFI and FFI experimental time, and resulting speed-up factor

	Mean time per injection run (sec.)			Estimated speed-up factor	
	RTL	Gate-level	FFI	FFI vs RTL	FFI vs Gate-level
MC8051	1.6	301.0	0.026	62	11577
AVR	2.0	72.0	0.026	77	2769
Microblaze	-	421.0	0.055	-	7655
NOELV	72.0	-	1.145	63	-

As it can be seen from the results in Fig.9, all dependability estimates obtained by BAFFI match with estimates obtained by RT-level and gate-level SFI, given that the discrepancy between them never exceeds the sampling error. This results not only support the validity of BAFFI results, but also indicate that BAFFI enables complex fault injection analysis of very large HW designs (like NOEL-V multicore CPU) which is usually unfeasible (or way too costly) to carry out by means of SFI. In fact, BAFFI provides similar fault injection precision (granularity) as simulation-based fault injectors, and at the same time reaches much higher experimental speed. Table I compares the observed experimental speeds of SFI and FFI setups. It can be seen that BAFFI accelerates fault injection in sequential logic (RT-level) by nearly two orders of magnitude, and accelerates fault injection in combinational logic (gate-level) up to four orders of magnitude. In such a way, by featuring both high FI precision and high experimental speed, BAFFI enables dependability analysis of fault-tolerant mechanisms in large SoCs.

Finally, it can be seen from the Table I that the board-side BAFFI setup (MC8051, AVR, Microblaze) on the average performs nearly 20x times faster than the host-controlled setup

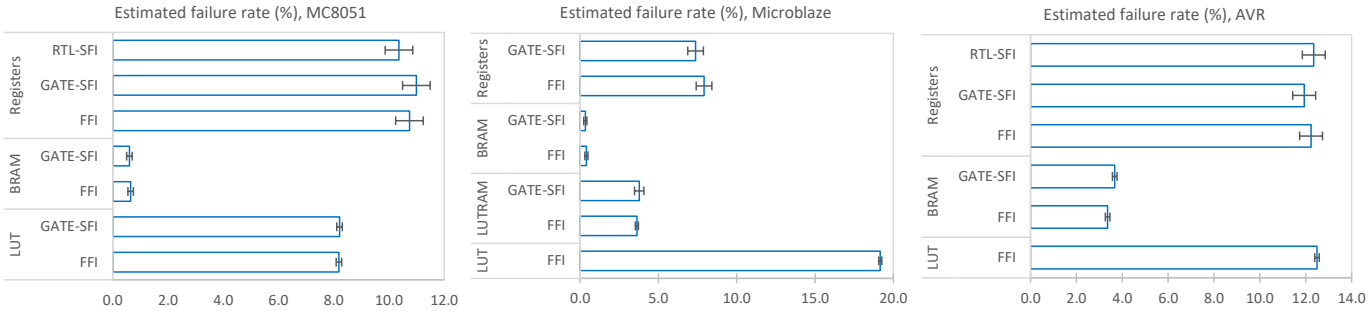

Fig. 9. Dependability estimates of three considered DUTs, obtained by RT-level SFI, gate-level SFI, and FFI experiments

(NOEL-V). This illustrates a speed-up gain attainable by eliminating slow (high-latency) synchronization/debug links with the host PC, and deploying FFI experiments in a fully embedded way in the target FPGA.

V. Conclusions

Dependability and security assessment of FPGA prototypes often requires a fine-grain FFI support. This paper has proposed a new FFI tool, that enables fault injection at the granularity of individual nestlist cells, such as registers, LUTs, BRAMs and LUTRAMs. This level of precision is achieved by locating with bit accuracy the essential bits of targeted netlist cells, and mapping them together under the hierarchical path in the DUT tree. Unlike the existing FFI tools, BAFFI does not introduce any area constraints (Pblocks) for targeted DUT components, thus featuring much lower intrusiveness of FFI process. Those FPGA resources that currently lack bit-accurate mapping rules (carry chains, DSPs and routing resources) are still targeted by BAFFI with the common Pblock granularity. This limitation will be addressed in the future work. BAFFI is published at https://gitlab.com/selene-riscv-platform/DAVOS, as an open-source extension to the DAVOS fault injection toolkit.

VI. Acknowledgment

This work has received funding from (i) ECSEL Joint Undertaking (JU) under grant agreement No 877056, (ii) Agencia Estatal de Investigación from Spain under grant agreement no. PCI2020-112092, (iii) European Unions Horizon 2020 research and innovation programme under grant agreement no. 871467, and (iv) Grant PID2020-120271RB-I00 funded by MCIN/AEI/ 10.13039/501100011033. Carles Hernández is partially supported by Spanish Ministry of Science, Innovation and Universities under "Ramón y Cajal", fellowship No. RYC2020-030685-I.

References

[1] D. de Andres, J. C. Ruiz, D. Gil, and P. Gil, "Fault emulation for dependability evaluation of vlsi systems," *IEEE transactions on VLSI systems*, vol. 16, no. 4, pp. 422–431, 2008.

[2] L. Antoni, R. Leveugle, and M. Feher, "Using run-time reconfiguration for fault injection in hardware prototypes," in *17th DFT Symposium*. IEEE, 2002, pp. 245–253.

[3] C. Lavin and A. Kaviani, "Rapidwright: Enabling custom crafted implementations for fpgas," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2018, pp. 133–140.

[4] L. A. Cardona and C. Ferrer, "Ac_icap: a flexible high speed icap controller," *International Journal of Reconfigurable Computing*, vol. 2015, p. 15, 2015.

[5] K. D. Pham, E. Horta, and D. Koch, "BITMAN: A tool and API for FPGA bitstream manipulations," in *DATE2017 Conference*. IEEE, 2017, pp. 894–897.

[6] R. Le, "Soft Error Mitigation Using Prioritized Essential Bits," 2012.

[7] A. Ramos, J. A. Maestro, and P. Reviriego, "Characterizing a risc-v sram-based fpga implementation against single event upsets using fault injection," *Microelectronics Reliability*, vol. 78, pp. 205–211, 2017.

[8] W. Yang, B. Du, C. He, and L. Sterpone, "Reliability assessment on 16 nm ultrascale+ mpsoc using fault injection and fault tree analysis," *Microelectronics Reliability*, vol. 120, p. 114122, 2021.

[9] L. A. Aranda, A. Sánchez-Macián, and J. A. Maestro, "Acme: A tool to improve configuration memory fault injection in sram-based fpgas," *IEEE Access*, vol. 7, pp. 128 153–128 161, 2019.

[10] A. Sari and M. Psarakis, "A fault injection platform for the analysis of soft error effects in fpga soft processors," in *19th DDECS Symposium*. IEEE, 2016, pp. 1–6.

[11] Stephanie Tapp, Xilinx Inc., "Configuration Readback Capture in UltraScale FPGAs, XAPP1230 (v1.1)," 2015.

[12] Xilinx Inc., "7 Series FPGAs Configuration UG470 (v1.13.1)," 2018.

[13] M. Jeong, J. Lee, E. Jung, Y. H. Kim, and K. Cho, "Extract lut logics from a downloaded bitstream data in fpga," in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2018, pp. 1–5.

[14] Xilinx Inc., "Soft Error Mitigation Controller v4.1," 2018.

[15] P. Swierczynski, G. T. Becker, A. Moradi, and C. Paar, "Bitstream fault injections (bifi)–automated fault attacks against sram-based fpgas," *IEEE Transactions on Computers*, vol. 67, no. 3, pp. 348–360, 2017.

[16] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert, "Statistical fault injection: Quantified error and confidence," in *Design, Automation and Test in Europe*, 2009, pp. 502–506.

[17] I. Tuzov, D. de Andrés, and J.-C. Ruiz, "Accurate robustness assessment of hdl models through iterative statistical fault injection," in *EDCC Conference*. IEEE, 2018, pp. 1–8.

[18] P. Tummeltshammer, "Analysis of common cause faults in dual core architectures," PhD dissertation, Technische Universitat Wien, 2009.

[19] C. Gaisler, *NOEL-V Processor*, 2020, https://www.gaisler.com/index.php/products/processors/noel-v.

[20] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović, "The RISC-V Instruction Set Manual," University of California, Berkeley, Tech. Rep. UCB/EECS-2014-54, May 2014.

[21] C. Gaisler, *GRMON3 User's Manual*, 2022.

[22] Oregano Systems GmbH, "MC8051 IP Core, User Guide (V 1.2)," 2013.

[23] J. Sauermann, "How to design your own CPU on FPGAs with VHDL," 2010. [Online]. Available: https://github.com/freecores/cpu_lecture

[24] Xilinx Inc., "Microblaze processor reference guide, ug984," 2019.

[25] I. Tuzov, D. de Andrés, and J.-C. Ruiz, "DAVOS: EDA toolkit for dependability assessment, verification, optimisation and selection of hardware models," in *DSN Conference*. IEEE, 2018, pp. 322–329.