Self-Test Library Generation for In-field Test of Path Delay faults

Lorena Anghel[†], Riccardo Cantoro[*], Riccardo Masante[*], Michele Portolan[‡], Sandro Sartoni[*], Matteo Sonza Reorda[*]

[*]Politecnico di Torino
Turin, Italy

[†]Univ Grenoble Alpes, CEA, CNRS, Grenoble INP
INAC-SPINTEC, 38000 Grenoble, France

[‡]Univ Grenoble Alpes, CNRS, Grenoble INP
TIMA, 38000 Grenoble, France

***Abstract*—New semiconductor technologies for advanced applications are more prone to defects and imperfections related, among several different causes, to the manufacturing process, aging and cross-talks. These phenomena negatively affect the circuit's timing and can be effectively modeled by means of the path delay fault (PDF) model. While path delay testing is currently supported by commercial Automatic Test Pattern Generation tools for scan designs, functional testing covering PDFs is not widely adopted, mainly because of the high cost for test generation. On the other side, functional test is already widely adopted for in-field test of stuck-at faults, which is often performed resorting to the execution of suitable test programs (Self Test Libraries, or STLs). This approach is attractive, since it can be performed at-speed with limited time constraints and high flexibility, making it a suitable in-field test solutions. Previous work assessed the feasibility and validity of functional approaches based on test programs targeting PDFs. In this work, we present the first systematic method for the development of very high fault coverage test programs for PDFs, which largely outperform test programs written for other fault models. Moreover, the proposed method allows the identification of functionally untestable faults. The effectiveness of the proposed approach was proven on an open-source RISC-V processor core, where 100% coverage of the functionally testable longest paths was achieved, compared with an initial coverage of 0.52% achieved with test programs targeting stuck-at faults. Results demonstrate that shorter paths are also effectively covered.**

***Index Terms*—software-based self-test, self test libraries, on-line test, path delay faults, safety**

I. Introduction

Several emerging applications increasingly require the use of advanced semiconductor technologies, due to their high computational capabilities in association with low power consumption. Such technologies use complex and sophisticated manufacturing processes, adopting advanced transistor designs, and allowing high operations frequencies and dense integrated circuit (IC) designs. However, these processes introduce new issues, including higher physical defects rates. Some of these defects can also arise during the device lifetime: in most cases they are related to ageing effects and overheating-related issues, making systems more sensitive to degradation than older generations. Other defects are also generated due to ElectroMagnetic Interference (EMI) or parasitic effects. For these reasons, testing the target device during the operation phase (*in-field test*) is becoming increasingly important, especially when the device is used in a safety-critical system. When dealing with the test (both at the end of manufacturing and in field) of ICs manufactured with the most advanced technologies, it is not possible to keep on relying on traditional fault models anymore, e.g., the stuck-at fault model, and efforts should be made towards replacing them with more advanced models, including delay faults, able to cover these newly identified issues. Transition and path delay faults (TDFs and PDFs, respectively), however, are not as popular and widely adopted as the stuck-at fault model, and solutions for dealing with TDFs and PDFs are not as mature. This poses new challenges for testing.

Current solutions to test delay faults are mainly based on Design for Testability (DfT) approaches, heavily relying on scan and *Built-In Self-Test* (BIST) circuitry. Such solutions have the advantage of being based on mature design and integration approaches currently supported by most commercial tools, providing also widely accepted coverage metrics. On the other side, DfT approaches require additional hardware, leading to an increase in area overhead and a decrease in the timing performance [1]. In addition to that, DfT solutions can lead to overtesting, since they test the circuit in a configuration which is different from the operational one. Finally, DfT solutions are rather invasive, since they require reconfiguring the circuit before running the test, hence destroying its current status. This may represent a critical point for in-field test. When the device under test (DUT) includes or is based on CPUs, functional solutions, mainly in the form of *Software-Based Self-Test* (SBST) [2], can be used to overcome the aforementioned issues. SBST approaches rely on forcing the CPU to execute a suitably crafted program, able to excite and propagate the effects of possible faults to some visible location. SBST solutions do not require the insertion of additional test structures (therefore, the DUT is not modified), they can be run at speed (which is a crucial aspect when dealing with delay faults), and they are particularly effective when performing in-field test, since they do not require any tester. Finally, they are less intrusive with respect to the system and the application. The test is conducted by executing a *Software Test Library* (STL) with purely functional stimuli applied to the DUT: in this way, "Functionally Untestable Faults" (FUFs) – i.e., faults whose effects never impact the DUT functionality – are never

detected, thus avoiding any form of overtesting. Given that a test based on STLs can be split in chunks of small duration and minimal invasiveness, the proposed methodology is especially well suited whenever in-field testing is required, allowing the system to be tested during idle time slots. It can be used to enhance DUT's safety throughout its operational lifetime, as required by standards like *ISO26262*. STLs are commonly provided by semiconductor companies, to be used on their CPUs or SoCs whenever these components are part of safety-critical systems that require comprehensive in-field testing [3]–[8]. Unfortunately, at the moment, functional testing for sequential circuits targeting path delay faults through an SBST approach lacks sufficient support from EDA tools, making it difficult to both develop the test and estimate its coverage. There are several works in the literature dealing with the functional test of generic circuits targeting path delay faults [9]–[16]. These works, however, are geared towards relatively simple circuits, usually taken from ISCAS workbenches and sometimes consisting of combinational circuits, only. As a consequence, none of the aforementioned works makes use of SBST solutions: for all these reasons, the scope of the presented work is quite different from that of the large majority of work in literature. The works [17], [18] describe techniques to test 100% of path delay faults on combinational circuits developed through a Reduced Order Binary Decision Diagram approach. Although effective, these works are not intended for in-the-field testing of complex sequential circuits such as processor cores. The article [19] describes an instruction-based self-test mechanism for pipelined processors targeting delay faults. This approach requires the device under test to be represented by means of graphs, hence leading to a high complexity when dealing with advanced processor cores. The authors in [20] present a methodology for the development of test programs for processor cores, focusing however on faults stemming from the datapath module only of non-pipelined CPUs.

Some works have investigated the topic of generating effective STLs for delay faults, specifically transition delay faults [21], [22]. The work in [23] represents the first step towards a functional test of PDFs by defining an appropriate test flow and analyzing the coverage achieved by STLs developed for stuck-at faults.

To the best of our knowledge, no strategies to develop STLs for path delay faults from all modules in modern pipelined CPUs are available to this day. This goal is particularly important today, given the widespread usage of STLs for in-field test of complex devices used in safety-critical applications, and the growing relevance of path delay faults to model aging effects. This paper proposes new solutions to generate STLs targeting PDFs and assessing the quality of the achieved results on a representative CPU core.

Our strategy is based on a two-steps approach. First, we generate test patterns for the combinational part of each target module within the DUT by using Automatic Test Pattern Generation (ATPG) methods. Patterns generated by the ATPG, however, might not be replicable by functional means only, as it might not be possible to generate them using the available instructions. For this reason, we provide functional constraints, so that patterns that cannot be replicated by functional means for a given CPU level are discarded. Generated patterns are then turned into CPU-related instructions by a semi-automatic procedure that involves automatic parsing and manual adjustments.

The main contributions of this paper are:

- A systematic and effective development methodology of PDF-oriented test programs for fully pipelined processor cores, leading to an important improvement with respect to the results achieved in [23],
- An effective test flow for functionally untestable faults identification,
- An experimental assessment of the effectiveness of the new method, which is shown to be able to detect 100.00% of the most critical paths and 87.31% of short paths on an open-hardware RISC-V based CPU.

To the best of our knowledge, this is the first work that tackles STL generation for path delay faults on complex pipelined CPUs, showing that it is possible to achieve very high results for PDF coverage on complex processor cores, targeting paths from the whole CPU.

The remaining of the paper is organized as follows: Section II outlines the background related to the path delay fault model, and the state-of-the-art works. Due to the lack of commercial solutions for path delay functional fault simulations, in Section III we introduce the fault simulation flow we devised, while in Section IV we present a systematic methodology for STL development capable of testing a generic processor core based on the fault simulation flow previously described. Section V gives details about the case study, and Section VI presents the achieved results. Section VII draws the conclusions.

II. Background

A. Path Delay fault model

Sequential circuits are systems where inputs and outputs are synchronized by a specific periodic clock signal. Such signal staggers the functioning of the whole circuit, by dividing it into clock cycles. For a sequential circuit to function correctly, all signals must keep a steady value for a certain time after a clock transition (hold time) and before the next clock transition (setup time), while transitions may occur within the clock cycle. Failures in satisfying these requirements may result in an erroneous logic behavior; the causes of such failures are known as *delay faults*. Combinational gates in a circuit are arranged into paths, from a "startpoint" to an "endpoint". Startpoints can either be primary inputs (PIs) or pseudo primary inputs (PPIs) – i.e., outputs of sequential elements. Similarly, primary outputs (POs) and pseudo primary outputs (PPOs) – inputs of sequential elements – are the endpoints. Critical paths are those paths whose slack – i.e., the difference between the clock period and the time it takes the signal to propagate through the whole path – is minimum. Path delay faults are used to

model physical defects, distributed along a path, that affect the nominal propagation delay of a given path. For each path, there are 2 PDFs: slow-to-rise (str), associated with a rising transition on the startpoint, and slow-to-fall (stf), associated with a falling transition on the startpoint. To test both path delay faults it is necessary to apply pairs of test vectors to the DUT at the maximum working frequency. Test vectors ensure that the correct transition is applied to any path's startpoint and is propagated to the endpoint by appropriately driving its off-path inputs, as shown in Fig. 1.

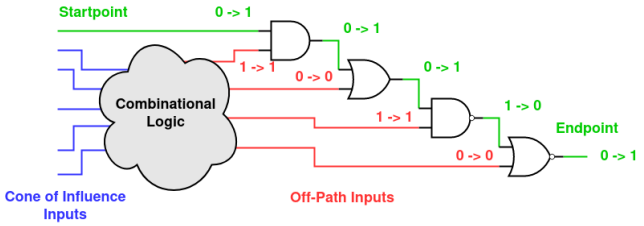

Fig. 1. Slow-to-Rise test vectors pair

The PDF model is currently supported by EDA tools through the adoption of Design for Testability techniques. Unfortunately, functional test of PDFs is not currently supported by any commercial fault simulation tool. For this reason, in this article we use the functional PDF test flow based on an SBST approach introduced in [23]. Faults are initially tested at a combinational level, by applying patterns recorded during the simulation of the test program execution by the DUT, obtaining a *combinational fault coverage*. Detected faults are then injected in the top-level netlist at the time of their detection in the previous step, and their effects are propagated through the sequential circuit to the POs. In this way, a *final fault coverage* is returned. A detailed explanation on the path delay functional fault simulation flow is given in Section III.

B. Related work

An exhaustive review of the state of the art on delay faults is given in [24]. An approach similar to our work has been followed in [25], where an ATPG is used to produce test instructions on a pipelined processor core targeting, however, stuck-at faults. Our work is complementary to [26], [27], in which the authors present strategies on how to insert monitors at the end of the most critical paths, so that slack violations are avoided, with a focus on aging effects. In [20], [28]–[31] methodologies for generating PDF-oriented test programs are presented. Works [20], [30] and [31] describe methods for generating test programs targeting specific computational blocks (e.g., datapath), both in non-pipelined and pipelined processors. Although results are significant, our work shows that testable critical paths can also be found in other modules, e.g., Control or Instruction Fetch Units, where the translation of patterns into pieces of code is much harder. In fact, testing paths from the latter modules is much harder with an SBST approach than those found in the datapath, requiring different testing strategies. Works [28], [29] propose a methodology for developing STLs based on evolutionary tools, targeting small non-pipelined processor cores. Unfortunately, such approach may not be easily extended to larger or more complex CPUs. Additionally, custom fault simulation tools are used [20], [28], [29]. Lastly, the authors of [19] present a test pattern generation method based on a graph representation of pipelined processors. However, creating such a representation can be a nontrivial task when dealing with advanced processor cores.

Works [9]–[12], on the other hand, focus on the generation of functional testing procedures through the use of scan chains and BIST modules. [9] presents a methodology to detect path delay faults by means of broadside (launch-on-capture) tests, in order to avoid overtesting due to path delay faults associated with long paths that cannot be sensitized during functional operations. Works [10], [11] discuss multicycle broadside tests applied to delay faults, with [11] focusing on transition delay fault diagnosis, showing relevant fault coverages. [12], finally, discusses a static test compaction procedure for test sets consisting of both broadside and skewed-load (launch-on-shift) tests, achieving a reduction in the test set size, as well as an increase in the fault coverage whenever a test set does not detect all the detectable faults. These works show that functional testing solutions for delay faults can achieve significant results. However, they require the usage of hardware structures to apply those tests, e.g., scan chains and BIST, while in this work we focus on the generation of test vectors to be converted into instructions belonging to STLs, without any other hardware support. Moreover, the previously aforementioned works are validated on small sequential circuits, while we focus on larger and more complex processor cores: generating instructions to test path delay faults is a much more complex task with respect to generating test vectors for sequential circuits.

III. PDF Functional Fault Simulation Flow

Path delay fault simulation is currently supported by commercial fault simulation tools only through the adoption of scan chains. Fault simulations for path delay faults in a scan circuit through commercial fault simulation tools is carried out as follows:

- Load a test vector through scan chains
- Apply a finite number (normally small) of functional clock cycles for the vector to propagate through the circuit and capture the response
- Download the response through scan chains, then repeat for all test vectors.

This mode is capable of fault simulating the circuit under test in a functional fashion, provided that such circuit is equipped with scan chains. The amount of functional clock cycles, however, is rather small compared to those needed by a typical STL; moreover, it requires the presence of DfT hardware. This, unfortunately, is not sufficient for a full sequential fault simulation targeting path delay faults. For this reason, prior to the description of the test program generation methodology, we present in detail the architecture of the flow we developed in order to perform path delay fault functional fault simulations.

This flow is devised such that, by providing the gate-level netlist of the DUT and a test program to be executed, the behaviour produced by each fault is evaluated. This allows identifying faults detected by the test program. Notice that, since this flow is devised for functional testing, the netlist should not use any scan chains. This implies that any fault is only observable, hence detectable, through at least one PO[1]. The STL generation methodology for the in-field testing of PDFs presented in this article is aimed towards modern pipelined processor cores, hence why from now onwards we will imply our DUT to be a generic pipelined CPU.

A schematic representation of the path delay fault simulation flow is given in figure 2. For the synthesized core, a list of paths is extracted by a Static Timing Analysis (STA) tool as well as the input stimuli obtained by performing a logic simulation of the test program. Then, the fault simulation process can be launched targeting path delay faults on the extracted paths. The fault simulation is divided into two steps, the first one is performed on the combinational modules of the DUT, the other one propagates faults observed at the combinational level to the POs through the pipeline stages of the sequential circuit.

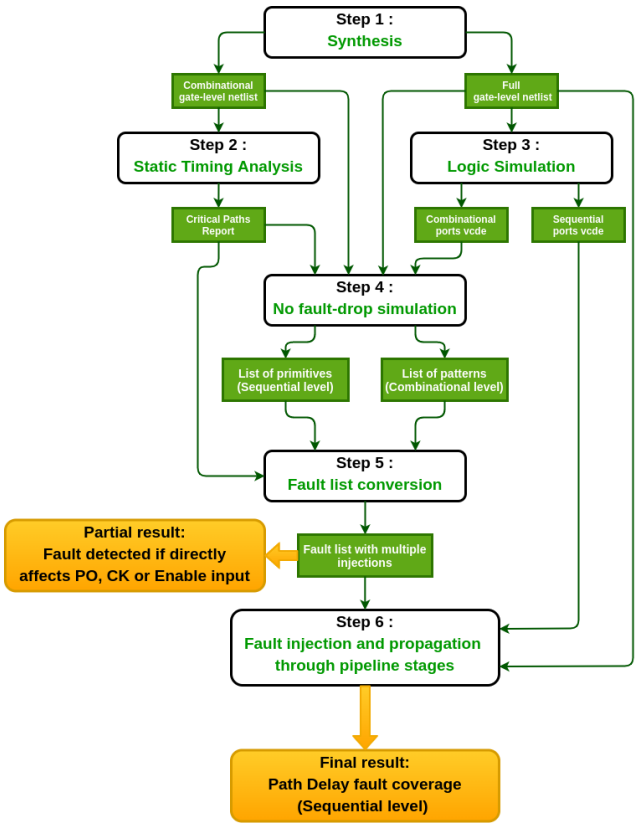

Fig. 2. Path delay test flow diagram

[1]Alternative observation mechanisms for test programs exist, such as checking the memory content at the end of the program execution [32], or observing the response of available safety mechanisms for in-field testing of safety-critical systems.

In the following subsections, a thorough explanation of all different steps is given.

A. Synthesis

The preliminary requirement to perform path delay fault simulations is to synthesize the target device such that the combinational cells of the gate-level netlist are grouped together separately from the sequential ones. In this way, we create a large submodule within the CPU that comprises all combinational cells; such submodule is then connected to sequential cells in the processor gate-level netlist.

Two netlists are then produced, one that is specific for the submodule holding all combinational cells, hereinafter referred to as *combinational netlist*, and one for the top-level module including both the combinational submodule and sequential cells, hereinafter referred to as *top-level netlist*. The output signals of the combinational netlist are either POs or PPOs, in case they are connected to the output signals of the top-level netlist or to inputs of sequential cells, respectively. We synthesize the circuit into these two netlists because in Section III-D the combinational netlist will be the module under test, while in Section III-E the top-level netlist will be needed.

B. Logic simulation

The second step consists in performing a logic simulation of test programs using any available logic simulation tools, with the main goal of generating input patterns for the subsequent fault simulation process. This step requires that the top-level netlist is instantiated as a component in the testbench, so that test program golden responses can be recorded both for combinational and top-level circuits. From now on, such golden responses are referred to as *patterns* lists. Such lists contain the value held by every combinational/top-level input and output port at any clock cycle during the execution of the test program, and will be used in Section III-D as test vectors for the fault simulation.

C. Static Timing Analysis

As a last preliminary step, the test flow generates the list of paths to be tested during the fault simulation. The standard approach consists in using a Static Timing Analysis tool to produce a list of combinational paths in ascending order of slack. In this way, in case the amount of extracted paths is too high, only the subset of combinational paths with the most stringent timing requirements is considered. In this regard, it is worth mentioning that STA tools are very pessimistic in performing their analysis and are not able to recognize *false paths*; such paths cannot be sensitized in the final design and would introduce untestable faults in the fault list. Therefore, a subsequent pruning of those paths from the initial path list is needed; this is only partially performed by commercial fault simulators as a preliminary phase of the fault simulation. As an example, the authors of [33] developed an algorithm to prune untestable paths, taking into account the circuit topology,

process variations, and aging effects; remarkably, this reduced the path count by 70.87%.

The benefits of refining the path list are non-trivial: if the list contains paths that cannot be tested by any means, the test engineer will fruitlessly try to activate and detect faults whose effects cannot be observed in any way, and the fault coverage will artificially drop. This is why we aim at improving the path extraction process, thus enhancing the standard STA-based methodology. To do so, we propose an iterative approach based on commercial Static Timing Analysis (STA) and ATPG tools, showed in figure Fig. 3.

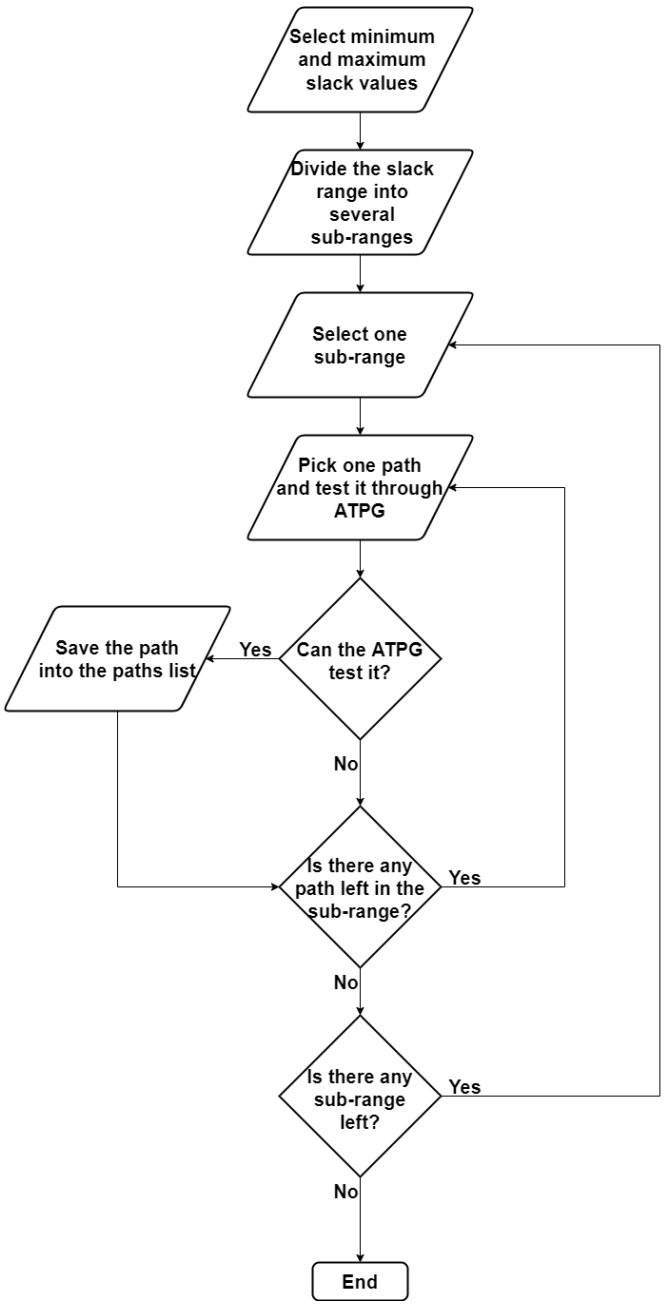

Fig. 3. Paths extraction flow

Iterating through several small slack intervals allows us to carefully pick the largest amount of testable paths throughout the whole slack range. In this way, we are sure that *structurally untestable faults* – i.e., faults that could not be tested even having full access to the input and output signals of the combinational block they are located in – are excluded. Not all structurally testable faults, however, are of interest when adopting an in-field test approach (e.g., for safety-critical applications), as some of them cannot be properly simulated or observed within functional scenarios. Such faults are known as *Functionally Untestable Faults* (FUFs). Consequently, it is crucial to identify as many functionally untestable faults as possible, removing them from the list of target faults to be considered.

The path list we generate is then used in the following fault simulation steps. Hereinafter, the set of produced paths is referred to as *path definition* list. Once the path definition list has been generated, it can be modified as needed: the proposed flow accepts any path definition list, hence every possible optimization or subset extraction is allowed.

When extracting paths for the subsequent fault simulation, timing information such as the slack associated to the path may be included as well. When performing path delay fault simulations, however, current commercial tools tend to neglect such timing information. For this reason, all data regarding slack or timing behaviors can be safely omitted in the path definition list.

D. Combinational-level fault simulation

Once the preparatory steps are cleared, the flow moves on to the fault simulation process, which is divided into two steps. In this first step, a fault simulation tool is used to feed, at any clock cycle, the input ports of the combinational netlist with patterns produced by the test program and stored in the Patterns list. This process allows us to identify all PDFs that produce a difference on at least a PO or a PPO when the considered test program is executed. Moreover, we identify the clock periods when this happens and the specific POs or PPOs affected by each fault. This information will be used at a later time on the top-level netlist. Fault simulators can also report, given any detected fault, which and how many patterns are able to detect it; however, for optimization reasons the simulators might "drop" (i.e. stop considering) some faults that are either already covered in other vector or that are not getting detected in a given time window. This analysis is accurate only when fault simulations are performed without fault dropping — that is, whenever a fault belonging to the active fault list is never dropped from it after being detected by any pattern.

The fault simulator reads the combinational netlist, the library files, the path definition list, and the combinational patterns list. More in detail, netlist and library files are used to build an internal model of the device under test, while the path definition list is analyzed to exclude false paths — hence, untestable faults — from the simulation. Lastly, signals included in the patterns list are interpreted as a list of pairs of vectors to be applied in sequence.

More accurate results in the overall flow can be achieved by running fault simulation without fault dropping; disabling fault dropping, however, significantly increases the fault simulation time. For this reason, by default, fault simulation tools drop every fault after being detected once. As previously mentioned, enabling the no fault-dropping option consists in never deleting faults, even when detected, from the active fault list. As a consequence, for each fault, we can obtain all patterns detecting it at the combinational level, instead of just the first one. This allows us to consider, in the following steps, the propagation of fault effects through the sequential logic not only for the first pattern, but for the whole test set.

E. Sequential-level fault simulation

Once we know which output of the combinational netlist is possibly affected by a certain fault at a specific time step or clock cycle, we have to propagate the fault effect throughout the sequential logic and check whether it reaches an observable point or the fault is vanishing. We modeled this sequential-level fault simulation by means of bit-flips injected in the sequential elements that capture the fault effect. For this purpose, we used a commercial tool different from what we have used in the previous step. The detected faults list obtained with previous strategies was made compatible with the tool used for the sequential-level fault simulation. In details, that means that each detected fault, together with its possible propagation endpoint and the time instant at which it reaches the sequential element, were translated into a bit-flip, applied to the faulty path endpoint at the aforementioned time instant.

It is worth noting that it is not obvious that each detected fault from the combinational circuits that could provoke a bit-flip can, in turn, be propagated to the POs. This is why, in the previous step, the fault-dropping option needs to be disabled: by allowing the generation of more patterns for each fault, it is also possible to generate several bit-flips at different time instants, hence increasing the accuracy of fault coverage evaluation. As a consequence, there may be more than one time instant at which a given fault is detected at combinational level.

Due to this reason, for each fault detected in the sequential-level fault simulation, the *minimum number of patterns* required to detect a fault can be defined. This value can be described in terms of the number of patterns generated by the functional fault simulation at the combinational level needed until the effect of the detected fault is propagated to the POs. The smaller this value is, the easier its test generation at sequential level is.

IV. STL Development Methodology

In this section, we thoroughly describe our STL development methodology, built on top of the previously presented fault simulation flow. Such STL development strategy is divided into three steps: functional constraint identification, test patterns generation using ATPG, and conversion of those patterns into instructions. This approach is summarized in Fig. 4.

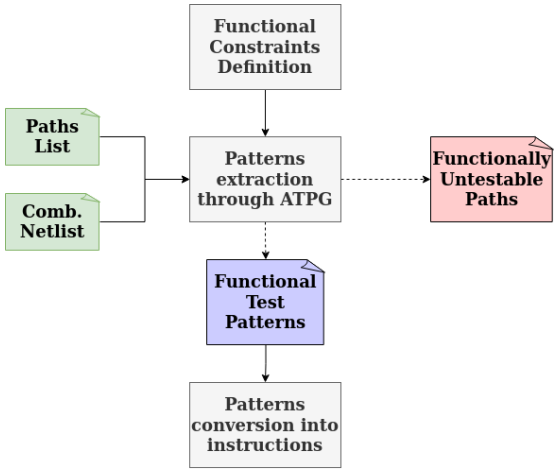

Fig. 4. Patterns generation flow

In the following, the three steps are described in detail.

A. ATPG pattern extraction

To test PDFs, test patterns must be able to generate and propagate specific transitions through the targeted paths. This task, however, cannot be fulfilled by any generic couple of vectors: very few patterns are capable of driving all PIs and PPIs properly. This is especially true when dealing with long paths: using random programs or even programs developed for other fault models, such as stuck-at faults (SAFs) or transition delay faults (TDFs), does not work effectively, and leads to a very small coverage. For this reason, special emphasis on the test pattern generation step should be placed.

The strategy we adopted for generating test vectors is summarized in Fig. 5.

The pattern generation task is managed by an ATPG, that requires the DUT's combinational netlist and the path list to produce the aforementioned test patterns. This allows producing effective and reliable patterns in a relatively short amount of time. It is noted, however, that the ATPG is not aware of the fact that these patterns have to be functionally applicable: this could lead to the generation of test patterns that cannot be translated into instructions. To mitigate this problem, we apply a set of functional constraints to the ATPG in order to promote the generation of test patterns that can be mapped into instructions belonging to the CPU's instruction set architecture (ISA). Such feature is described thoroughly in Section IV-B.

Faults managed by the ATPG following this approach may belong to one of the following categories:

- *Detected* group: the fault has been detected and the relative couple of test vectors have been produced,
- *ATPG Untestable* group: the ATPG could not generate test vectors capable of testing the fault under the specified constraints.

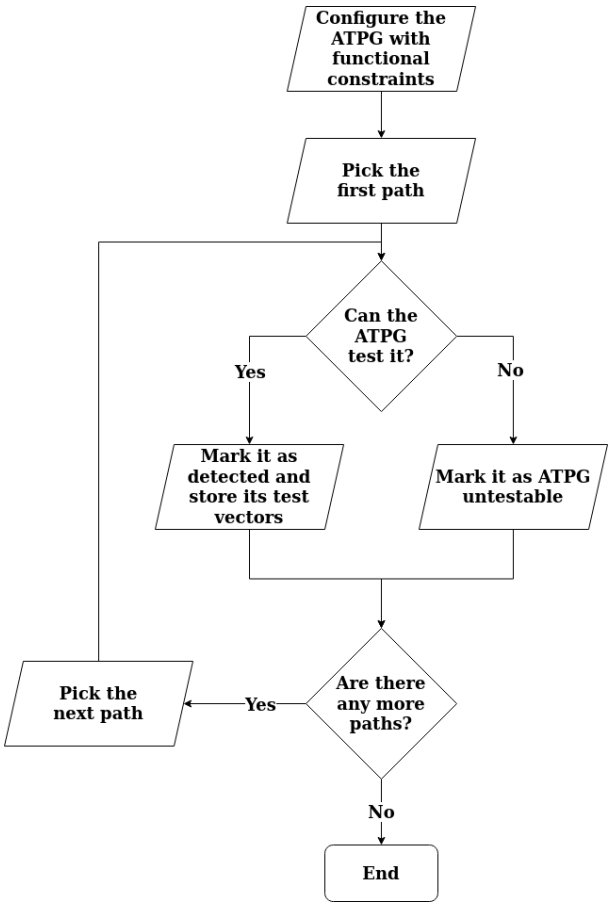

Fig. 5. ATPG-based test vectors generation

The latter category requires some observations. Based on what we presented in Section III-C, faults produced by our flow are known to be testable by the ATPG when no constraint is applied. As a consequence, if a fault is marked as ATPG untestable it means that this fault is also functionally untestable, as the ATPG cannot produce test vectors when functional constraints are applied. In this way, we are able to remove most FUFs from the fault list, obtaining a list of faults whose patterns can be converted into instructions. Nevertheless, it is still possible that there might be a small subset of functionally untestable faults among the detected ones. The reason for this lies in the architecture of the DUT, but also on how instructions are issued and executed by the processor core. To give an example, let us consider a fault that affects the arithmetic unit of a pipelined, in-order processor and the test pair produced by the ATPG requires launching a division followed by another ALU operation. Given this premise, the targeted fault is a FUF: divisions require more than one clock cycle to complete, and their execution stalls the whole CPU, hence it will be impossible to execute an ALU operation at the clock cycle following the issuing of the division. It is worth reiterating, however, that functionally untestable faults cannot be detected by SBST means, due to the fact that they cannot be excited under functional scenarios.

B. Functional constraints identification

Functional constraints are a crucial component of the ATPG pattern extraction process, as they ensure that the produced test vectors can be effectively translated into instructions belonging to the DUT's instruction set architecture (ISA). For this reason, identifying and defining a set of functional constraints is a key task in our methodology, and is presented separately.

The problem of producing instruction sequences for CPUs is well-known in academia, with several works in literature that tackle it. [34] presents a technique to generate instructions to test delay faults on pipelined processor cores. This is done in three steps, namely an ATPG-based delay test generator to be applied to the combinational portion of the device under test, a verification engine based instruction mapper, and a feedback mechanism. The generation of functional instructions to test delay fault is done by means of a set of Verilog properties to be fed to a bounded model checker. This, however, can be quite taxing, as properties are built looking at inputs of paths to be tested, leading to a large number of properties to be fed to the bounded model checker. Moreover, writing all these properties requires a non-negligible amount of manual work. In this paper, we propose a semi-automatic methodology to developing functional constraints, capable of identifies all inputs that cannot be controlled once sub-modules of the DUT are provided. [35] proposes an algorithm, based on formal techniques, that takes the gate-level description of a pipelined processor as input and generates a sequence of assembly instructions able to stress any module within it by maximizing the switching activity. Although effective, this methodology is geared towards the generation of stress-oriented assembly instructions, while our goal is to excite and propagate faults affecting paths inside CPUs. In addition, formal techniques may require a non-negligible amount of time. [36] introduces an approach to generate instruction sequences for SBST, and makes use of a Validity Checker Module to limit test sequences to valid RISC-V instructions and the given environment. This approach deals with the well-known stuck-at fault model, which is quite different with respect to the path delay fault one. Moreover, for larger circuits it was not able to complete the generation of SBST-based routines. Finally, [37] proposes an approach for the automatic generation of SBST programs. This is done by generating functional test sequences through the usage of an ATPG, followed by the adoption of a validity checker module (VCM) which allows the specification of constraints with regard to the generated sequences. The VCM is the used to express typical constraints that exist when SBST is adopted for in-field test. This approach shows that it is possible to automatically develop effective test programs for in-field testing. The fault model targeted in [37], however, is the stuck-at fault one, while in this paper we focus on path delay faults.

The functional constraint identification algorithm is briefly summarized in Algorithm 1.

In this work, the set of functional constraints needed to generate valid test patterns is directly applied to the ATPG.

Algorithm 1: Functional Constraint Identification

```
Input  : A set S:=(S_i), where S_i is a sub-module of the
         device under test
Input  : A set D:=(P_i, C_i), where P_i is a path to be tested
         and C_i is the list of the i-th path's input cone logic
Output: A list of functional constraints to be applied to the
         ATPG
begin
    F := empty list of functional constraints
    /* Repeat for every sub-module                    */
    foreach S_i in S do
        NC:= empty set of non-controllable input signals;
        select all paths P_i belonging to S_i;
        extract all C_i related to paths P_i;
        run logic simulation of ad-hoc programs;
        store non-controllable input signals into NC;
        /* Check whether non controllabe
           inputs belong to the C_i of a P_i
           within the considered S_i                   */
        foreach signal in NC do
            if signal in C_i then
                add signal with its tied value to F;
            end
        end
    end
    return F
end
```

Such constraints come in the form of values applied to the PIs and PPIs of the DUT: for this reason, prior to any further step, we perform an *off-path inputs analysis*, i.e., for each path we analyze the path's input cone logic, starting from off-path inputs and moving towards the *cone of influence inputs*. This, together with a list of sub-modules into which the DUT is divided, are the input of our functional constraint identification algorithm.

Identifying the values to be applied to the input pins is not an easy task. We devised a semi-automatic approach to tackle this problem. For each sub-module, first we identify all paths within the sub-module and the input cone logic for each path. Next, we employ an automatic tool capable of performing logic simulations of carefully devised programs. Each program consists of every possible combination of instructions that affect the targeted sub-module: for instance, if we take into account the LoadStore unit, one given program will contain all possible combinations of *load* and *store* instructions. While simulating, our automatic tool records all input values. Once all simulations are completed, it identifies all the input pins that cannot be controlled when running test programs. Once this is cleared, the last step consists of annotating the set of input signals, together with their value, into a list of constraints to be fed to the ATPG. In this way, we make sure that the produced test patterns only involve those pins that can be driven through SBST means with replicable values.

C. Pattern conversion

Once the test pairs have been generated, it is necessary to translate them into instructions. The very first step in performing such conversion consists of mapping the values stored in the test patterns into signal groups, e.g., the *opcode*, *ALU operands*, *registers*, etc. Afterwards, the test engineer must choose a set of instructions that is capable of replicating the generated test pair. This process is quite complex for two reasons. First, for each available instruction, it is necessary to understand which signals are controllable and how the instruction affects them; this could be non-trivial depending on the signal group. The second reason is due to the fact that values from the test vector must be applied concurrently on CPU pipeline stages. As a consequence, a single vector has to be mapped to several instructions, each one controlling signals in one pipeline stage. This has to be carefully selected such that they reproduce a test vector in a given clock cycle. In our methodology, this is done employing a semi-automated approach: a parser maps the test vectors into the aforementioned signal groups, while the test engineer defines a sequence of instructions and simulates them to make sure that they reproduce the required values.

A generic sequence of instruction is divided into three blocks:

1) Initialization instructions: mainly in the form of *load* instructions, they are used to initialize the DUT so that the test can be effectively applied.
2) Test instructions: once the DUT is prepared, these instructions generate and propagate the intended transition through the targeted path.
3) Store instructions: to propagate data affected by errors to POs, where mismatches due to faults are finally observed.

To better clarify, in the following we report an example of pattern conversion of a real test vector couple, depicted in a simplified version in Fig. 6. To give some reference, the path targeted by the ATPG in the figure belongs to an adder embedded in the jump address sub-module. To start off, in the upper table we reported a subset of the two test vectors, namely *First Vector* and *Second Vector*, whose values have already been mapped into their respective signal groups. More in details, we highlighted signals that could be easily identified within modules of the processor core: in this example, registers from the register file are showed in the upper part, while portions of the fetched instruction are showed in the lower part. The dash symbol, '-', has been used to represent the "don't care" value, while in all other cases hexadecimal values are reported. In this specific example, we decided to split the instruction into several fields:

- id_instruction[6:0]: contains the opcode of the instruction (bold black text),
- id_instruction[14:12]: contains a special function field (func3) or the lower portion of the immediate field depending on the instruction (bold pink text),
- id_instruction[19:15]: contains the register source 1 field or a portion of the immediate field depending on the instruction (bold green text),
- id_instruction[24:20]: contains the register source 2 field or a portion of the immediate field depending on the

instruction (bold red text),

- id_instruction[31:25]: contains a special function field (func7) or the upper portion of the immediate field depending on the instruction (bold brown text).

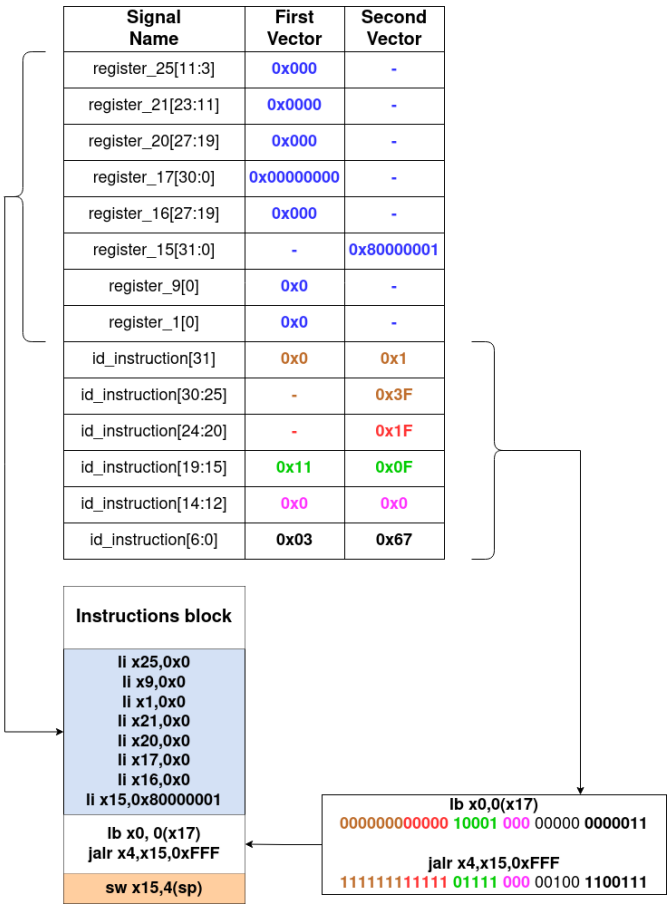

Fig. 6. Example of test program

Next, we look for instructions that can match the required transitions, following the model previously discussed: such instructions are grouped within the *Instructions block* box.

This code snippet is divided into three sections, reflecting the previously described blocks. Starting with the first one, highlighted in light blue, it is possible to see the set of initialization instructions, consisting of all *load immediate* (*li*) that set the registers to their appropriate value. Following, the two instructions responsible for the generation of the required transition are reported in the white block below. The first instruction serves the purpose of initializing the path to be tested with the appropriate set of values, while the second triggers the transition so that it propagates through the path towards its endpoints. Finally, we introduce a store instruction to propagate the effects of the previously excited faults towards one of the POs, hence observing data affected by the targeted path delay fault. This is done so that the faulty value can be stored into a non-volatile memory, possibly after being compacted into a signature, to be compared against the golden circuit's one. In this sense, observing a faulty value at the primary output equals to detecting its relative fault. Choosing the right instructions is not trivial as they must be carefully picked to replicate all the signals in the test vectors without any mismatch, bearing into mind how they behave across all pipeline stages.

Finally, the STL consists of every sequence of instructions generated for each path. One of the strengths of this approach consists in being modular: depending on the length of the test slot, i.e., the time slot in which the DUT can be tested, the test engineer can decide whether to run the test program as a whole, or split it into several submodules. The only rule to be observed is not to split a single instruction block that must always be run as a whole, since the ability of testing a path strictly depends on the execution of the instruction sequence without interruptions.

V. Case Study

A. Processor core

The adopted DUT is an open hardware single-core SoC platform centered on PULPino, a 32-bit RISC-V core developed by ETH Zurich and Università di Bologna [38]. In our work, PULPino was configured to use the RI5CY core, an in-order, single-issue core with 4 pipeline stages, capable of supporting the RV32ICM instruction set which includes integer, compressed, and multiplication instructions. Its schematic representation is showed in Fig. 7.

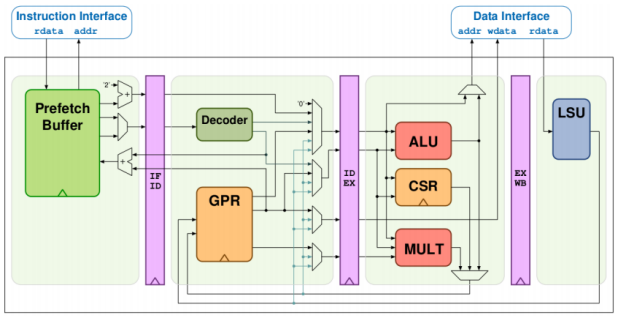

Fig. 7. PULPino Core Representation

This processor core was synthesized using the open hardware 45nm *Nangate OpenCell Library* provided by Silvaco [39]. Data regarding the synthesized core can be found in Table I.

TABLE I
Case Study General Info

Parameter	*Value*
Number of gates	46,850
Total Area (eq. gates)	51,001.65
Clock Period (ns)	5.00

Table II reports additional information on the extracted paths. These paths are the result of the extraction process described in Section III-C; as a consequence, only paths that can be tested through ATPG are reported here and targeted

in our fault simulation experiments. Two types of paths were extracted: long paths, whose slack is small, and short paths, with a larger slack. Paths from both these categories have been targeted in our fault simulation experiments.

TABLE II
PATHS REPORT

Paths class	*#Paths extracted*	*Slack range (ns)*
Long paths	5,009	[1.3 : 2.5]
Short paths	5,137	[4.8 : 4.97]

Longer paths have been extracted by looking at all paths in the slack range $[0 : 2.5]$ns: the reason for not having any path in the $[0 : 1.3]$ns slack range lies in the fact that the ATPG did not find any testable path among those. From a topological point of view, long paths are divided into three groups:

1) 45 paths in an adder of the divider module of the ALU,
2) 540 paths in an adder of the load/store unit,
3) $4,424$ paths in an adder inside the jump address module of the decode stage.

Shorter paths, on the other hand, have been limited to those belonging to the $[4.8 : 5.0]$ns slack range. As there are plenty of short paths in a circuit, the extracted ones do not belong to single modules of the CPU, rather they are scattered throughout the whole processor core.

B. Test programs

Since we are working with two different classes of paths – long paths and short paths – we decided to develop two separate STLs.

Starting with the long paths STL, this set of procedures was developed completely from scratch, and it can be thought of as three independent procedures that aim at testing faults in the three aforementioned submodules. For the vast majority of faults, our methodology was capable of producing appropriate testing instructions that excite faults and propagate their effects towards primary outputs. For a very small percentage of all faults (45 out of $10,018$, to be found in the divider unit), however, the test vectors could not be replicated due to physical constraints: such couples required a division promptly followed by another ALU-related instruction, which cannot be executed at the immediately following clock cycle as divisions take more than one clock cycle to complete, making those 45 faults functionally untestable.

As for the short paths STL, experimental data from [23] shows that programs developed for SAFs and TDFs are somehow capable, although not with high coverages, to test short paths. For this reason, we decided to start from the already available SAF and TDF-oriented programs. In this way, we could discard the faults that are already covered by them, and focus on the remaining ones through our proposed methodology. Thanks to this approach, we were able to easily identify a large number of FUFs, as well as to further increase the number of detected faults. These results validate our approach, as it succeeded in producing effective STLs even though long and short paths are quite different in terms of topological distribution and functionalities.

VI. EXPERIMENTAL RESULTS

The experimental results reported in this article have been obtained through the adoption of several different commercial tools coordinated through the usage of Bash and Python scripts as described in Section III. Regarding the preliminary steps, we used *Design Vision*, provided by *Synopsys*, for the synthesis step, together with *Questasim* by *Mentor Graphics* for logic simulations and *PrimeTime* for path extraction in conjunction with *TetraMAX*, both by *Synopsys*, for path refinement. The combinational-level fault simulation has been performed using *TetraMAX*, while the sequential-level fault simulation makes use of *Z01X*, a fault simulator specifically designed for functional safety by *Synopsys*.

The experimental results we gathered are referred to the processor core and synthesis library described in Section V, and are reported separately for long and short paths. The STLs we generated were capable of covering 100% of testable long paths and 87.31% of testable short paths of the chosen DUT. In both cases, the STL generation required about 8 days of ATPG time, using 5 cores of an Intel Xeon CPU E5-2680 v3 server.

Starting with the long paths STL, the whole test program requires 26kB of memory space, and a total amount of $45,605$ clock cycles to execute. The amount of clock cycles has been computed by executing the whole STL in a single simulation; depending on the situation, the test engineer can then split the STL into several shorter testing sub-routines to adapt them to the available idle time slots, matching the strict time constraints of in-field test. Since no methodologies to develop STLs for all PDFs in a processor core are currently available in literature, to assess the validity of the proposed methodology we decided to compute the fault coverage figures for STLs intended for other fault models – namely, SAFs and TDFs – used as a reference case. Stuck-at fault oriented test vectors are not compatible with delay faults when generating test patterns for scan chains through ATPG. Nevertheless, in this article we adopt the SBST paradigm, where test vectors are applied through functional at-speed clock cycles throughout the whole STL duration. For this reason, it is appropriate to compare our results with those achieved by SAF-oriented test programs. In particular, we run 5 different SAF programs and 1 TDF program, able to achieve 90.91% (cumulatively) and 74.25% wrt their target faults, respectively. All these data are reported in Table III. The *Combinational Fault Coverage* column shows the path delay FC achieved on the combinational portion of the DUT while *Final Fault Coverage* reports the fault coverage for the whole CPU. The last row reports information about the test program generated by our method.

The table clearly shows that our method dramatically improves the FC figures of existing test programs. The reason for this significant difference is that in our test program's instructions and values are carefully chosen to test specific paths; other test programs, instead, can be approximated to a

TABLE III
LONG PATH FAULT COVERAGE

Program	*#Clock Cycles*	*Combinational Fault Coverage*	*Final Fault Coverage*
SAF Program 1	64,502	0.33%	0.32%
SAF Program 2	36,394	0.27%	0.27%
SAF Program 3	42,970	0.27%	0.22%
SAF Program 4	118,098	0.40%	0.32%
SAF Program 5	17,269	0.09%	0.08%
TDF Program	23,451	0.27%	0.23%
PDF Program	45,605	99.50%	99.50%

random – and thus ineffective – approach due to the significant differences between the three fault models. We also analyzed the FC figures on the three submodules of the CPU into which the longest paths are found, as reported in Table IV.

TABLE IV
LONG PATH FAULT COVERAGE PER MODULE

Module	*#Faults*	*Combinational Fault Coverage*	*Final Fault Coverage*	*Testable Fault Coverage*
ALU_Div Adder	90	50.00%	50.00%	100.00%
LoadStore Adder	1,080	100.00%	100.00%	100.00%
Jump_Addr Adder	8,848	100.00%	100.00%	100.00%
Total	10,018	99.50%	99.50%	100.00%

We achieved a 100% FC in both the load/store unit and the jump address adder, while we were able to cover 45 out of the 90 faults of the divider. The remaining 45 faults have been identified as FUFs, as previously explained. Consequently, if we exclude such FUFs from the final fault coverage, we obtain a testable fault coverage for all three modules – and, thus, for all long paths – equal to 100%.

Similarly, Table V reports data for the STL developed for short paths (last row). This program requires 18kB of memory space and a total amount of $279,253$ clock cycles to execute completely. Since short paths are distributed among the whole CPU, we decided to associate them to the module, or set of PIs, from which the path's startpoint stems, also reporting the relative FC. Table VI reports such data.

TABLE V
SHORT PATH FAULT COVERAGE

Program	*#Clock Cycles*	*Combinational Fault Coverage*	*Final Fault Coverage*
SAF Program 1	64,502	71.98%	50.10%
SAF Program 2	36,394	73.76%	58.00%
SAF Program 3	42,970	74.78%	55.70%
SAF Program 4	118,098	76.51%	67.60%
SAF Program 5	17,269	73.16%	51.25%
TDF Program	23,451	73.60%	56.70%
PDF Program	279,253	83.77%	77.15%

For each module, we show the total amount of faults, the combinational and final fault coverages, as well as the testable fault coverage achieved by removing FUFs from the final FC. Most paths belong to the *ID_Stage*, which is well

TABLE VI
SHORT PATH FAULTS COVERAGE PER MODULE

Startpoint	*#Faults*	*Combinational Fault Coverage*	*Final Fault Coverage*	*Testable Fault Coverage*
Debug_PIs	666	0.00%	0.00%	100.00%
Other_PIs	299	92.31%	81.60%	81.60%
CS_Registers	290	69.56%	34.48%	41.38%
Debug_Module	287	0.00%	0.00%	100.00%
ID_Stage	7,714	97.54%	94.28%	94.82%
Controller	80	0.00%	0.00%	50.00%
Pipeline_Regs	744	94.35%	69.22%	69.49%
Registers	6,890	99.01%	98.08%	98.08%
EX_Stage	652	57.05%	22.38%	22.65%
ALU	182	11.53%	11.53%	17.37%
Multiplier	48	33.33%	33.33%	43.33%
Sparse_logic	422	79.38%	25.82%	25.82%
LoadStore_Unit	366	81.15%	81.15%	84.70%
Total	10,274	83.77%	77.15%	87.31%

covered by our STL. Two blocks of faults are marked as completely untestable, namely those that stem from debug-related circuitry, as they are not controllable by functional programs. Faults that originate from non-debug PIs are easily controllable – hence a 92.31% combinational coverage – but not always easily propagated to primary outputs, with a final 81.60% coverage. Moving to faults related to *CS_Registers*, 20 of them are also affected by clock gating circuitry and, hence, untestable. As for faults belonging to the *EX_Stage*, in most cases they are found in control-related logic, making it hard to either control or observe them. Lastly, looking at the *LoadStore_Unit*, some faults could not be properly excited due to the presence of off-path inputs from other modules and PIs; observability however is quite easy to achieve, as every fault detected at the combinational level is also observed at the POs.

As a final note, we would like to highlight the fact that the fault list we produced and used throughout all fault simulation steps is a generic one, not depending on the application the DUT is executing. This detail is relevant as, based on the application, some working modes, and hence circuitry, of the DUT might never be used. Consequently, faults located in the unused circuitry can be effectively marked as Safe Faults (i.e., FUFs), as per ISO26262 standard. The results achieved in this work can thus be enhanced by a careful analysis of those working modes that we can neglect, e.g., identifying a set of constraints to be fed to a formal verification tool that identifies the aforementioned FUFs, as described in [40].

VII. CONCLUSIONS

In this paper we presented a systematic STL generation methodology targeting path delay faults in fully pipelined CPUs and compared its results with those obtained by using test programs developed for other fault models. Even though this approach is still not fully automated, thanks to it, we were able to cover 100% of all testable long paths, which were very poorly tested by other test programs, as well as to enhance the coverage of short paths by identifying large portions of untestable paths as well as testing some more, reaching a final 87.31% coverage.

To the author's best knowledge, this is the very first approach that allows a semi-automated and deterministic approach to functional test generation for PDFs by leveraging industrial ATPG capabilities. Moreover, its flexibility and limited runtime make the method an ideal candidate for the on-line periodical tests mandated by safety standards such as ISO 26262.

Future works will focus on the inclusion of aging effects, a major contributor to in-field delay faults. In this context, our approach will be the basis to create STL routines able to detect aging-induced PDFs without the need of embedded hardware monitors.

REFERENCES

[1] A. D. Singh, "Scan based two-pattern tests: should they target opens instead of tdfs?" in *16th IEEE Latin-American Test Symposium (LATS)*, March 2015, pp. 1–2.

[2] M. Psarakis *et al.*, "Microprocessor software-based self-testing," *IEEE Design Test of Computers*, vol. 27, no. 3, pp. 4–19, 2010.

[3] Hitex, "Microcontroller self-test libraries." [Online]. Available: https://www.hitex.com/tools-components/software-components/selftest-libraries-safety-libs/pro-sil-safetlib/

[4] STMicroelectronics, "Guidelines for obtaining IEC 60335 Class B certification for any STM32 application," Mar 2016. [Online]. Available: http://www.st.com/content/ccc/resource/technical/document/application_note/02/1a/91/78/e4/15/4d/35/CD00290100.pdf/files/CD00290100.pdf/jcr:content/translations/en.CD00290100.pdf

[5] Cypress Semiconductor, "FM3 and FM4 Family, IEC61508 SIL2 Self-Test Library." [Online]. Available: https://www.cypress.com/file/249196/download

[6] Renesas Electronics, "SSP Supplemental Add-Ons." [Online]. Available: https://www.renesas.com/en-eu/products/synergy/software/add-ons.html

[7] Microchip Technology Inc., "16-bit CPU Self-Test Library User's Guide," 2012. [Online]. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/52076a.pdf

[8] ARM, "Enabling Our Partnership to Bring Safer Solutions to the Market Faster." [Online]. Available: https://developer.arm.com/technologies/functional-safety

[9] I. Pomeranz, "On the detection of path delay faults by functional broadside tests," in *2012 17th IEEE European Test Symposium (ETS)*, 2012, pp. 1–6.

[10] ——, "Generation of multi-cycle broadside tests," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 8, pp. 1253–1257, 2011.

[11] ——, "On transition fault diagnosis using multicycle at-speed broadside tests," in *2011 Sixteenth IEEE European Test Symposium*, 2011, pp. 189–194.

[12] ——, "Static test compaction for delay fault test sets consisting of broadside and skewed-load tests," in *29th VLSI Test Symposium*, 2011, pp. 84–89.

[13] M. Srinivas *et al.*, "Functional test generation for path delay faults," in *Proceedings of the Fourth Asian Test Symposium*, 1995, pp. 339–345.

[14] M. Micheal *et al.*, "Atpg for path delay faults without path enumeration," in *Proceedings of the IEEE 2001. 2nd International Symposium on Quality Electronic Design*, 2001, pp. 384–389.

[15] I. Pomeranz *et al.*, "Vector-based functional fault models for delay faults," in *Proceedings Eighth Asian Test Symposium (ATS'99)*, 1999, pp. 41–46.

[16] M.-T. Hsieh *et al.*, "High quality pattern generation for delay defects with functional sensitized paths," in *2008 17th Asian Test Symposium*, 2008, pp. 131–136.

[17] A. Matrosova *et al.*, "Pdf testability of the circuits derived by special covering robdds with gates," in *East-West Design & Test Symposium (EWDTS 2013)*, 2013, pp. 1–5.

[18] T. Shah *et al.*, "Test pattern generation to detect multiple faults in robdd based combinational circuits," in *2017 IEEE 23rd International Symposium on On-Line Testing and Robust System Design (IOLTS)*, 2017, pp. 211–212.

[19] V. Singh *et al.*, "Instruction-based self-testing of delay faults in pipelined processors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 11, pp. 1203–1215, Nov 2006.

[20] Wei-Cheng Lai *et al.*, "Test program synthesis for path delay faults in microprocessor cores," in *IEEE Intl. Test Conference*, 2000, pp. 1080–1089.

[21] R. Cantoro *et al.*, "Self-test libraries analysis for pipelined processors transition fault coverage improvement," in *2021 IEEE 27th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, 2021, pp. 1–4.

[22] ——, "Effective techniques for automatically improving the transition delay fault coverage of self-test libraries," in *2022 IEEE European Test Symposium (ETS)*, 2022 [In press].

[23] R. Cantoro *et al.*, "New perspectives on core in-field path delay test," in *2020 IEEE International Test Conference (ITC)*, 2020, pp. 1–5.

[24] J. Mahmod *et al.*, "Special session: Delay fault testing - present and future," in *2019 IEEE 37th VLSI Test Symposium (VTS)*, 2019, pp. 1–10.

[25] S. Gurumurthy *et al.*, "Automatic generation of instruction sequences targeting hard-to-detect structural faults in a processor," in *2006 IEEE Intl. Test Conference*, 2006, pp. 1–9.

[26] Z. Ghaderi *et al.*, "Sensible: A highly scalable sensor design for path-based age monitoring in fpgas," *IEEE Transactions on Computers*, vol. 66, no. 5, pp. 919–926, 2017.

[27] A. Sivadasan *et al.*, "Nbti aged cell rejuvenation with back biasing and resulting critical path reordering for digital circuits in 28nm fdsoi," in *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2018, pp. 997–998.

[28] P. Bernardi *et al.*, "On the automatic generation of test programs for path-delay faults in microprocessor cores," in *12th IEEE European Test Symposium (ETS'07)*, May 2007, pp. 179–184.

[29] K. Christou *et al.*, "A novel sbst generation technique for path-delay faults in microprocessors exploiting gate- and rt-level descriptions," in *26th IEEE VLSI Test Symposium (vts 2008)*, April 2008, pp. 389–394.

[30] N. Hage *et al.*, "On testing of superscalar processors in functional mode for delay faults," in *30th IEEE Intl. Conference on VLSI Design and 16th IEEE Intl. Conference on Embedded Systems (VLSID)*, 2017, pp. 397–402.

[31] C. H. . Wen *et al.*, "On a software-based self-test methodology and its application," in *23rd IEEE VLSI Test Symposium (VTS'05)*, 2005, pp. 107–113.

[32] J. Perez Acle *et al.*, "Observability solutions for in-field functional test of processor-based systems: A survey and quantitative test case evaluation," *Microprocessors and Microsystems*, vol. 47, pp. 392 – 403, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0141933116301867

[33] J. Chen *et al.*, "Identification of testable representative paths for low-cost verification of circuit performance during manufacturing and in-field tests," in *32nd IEEE VLSI Test Symposium (VTS)*, April 2014, pp. 1–6.

[34] S. Gurumurthy *et al.*, "Automatic generation of instructions to robustly test delay defects in processors," in *12th IEEE European Test Symposium (ETS'07)*, 2007, pp. 173–178.

[35] N. I. Deligiannis *et al.*, "Effective sat-based solutions for generating functional sequences maximizing the sustained switching activity in a pipelined processor," in *2021 IEEE 30th Asian Test Symposium (ATS)*, 2021, pp. 73–78.

[36] T. Faller *et al.*, "Towards sat-based sbst generation for risc-v cores," in *2021 IEEE 22nd Latin American Test Symposium (LATS)*, 2021, pp. 1–2.

[37] A. Riefert *et al.*, "A flexible framework for the automatic generation of sbst programs," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 10, pp. 3055–3066, 2016.

[38] ETH Zurich and Università di Bologna, "PULPino microcontroller system." [Online]. Available: https://github.com/pulp-platform/pulpino

[39] Silvaco, "Silvaco 45nm open cell library." [Online]. Available: https://si2.org/open-cell-library/

[40] F. A. da Silva *et al.*, "Determined-Safe Faults Identification: A step towards ISO26262 hardware compliant designs," in *IEEE European Test Symposium (ETS)*, 2020, pp. 1–6.

Lorena ANGHEL is Full Professor at Grenoble INP and member of the research staff of SPINTEC Laboratory. She received the PhD from Grenoble INP in 2000, Cum Laudae. Her research interests include hardware design and test of neural networks, on-line testing, fault tolerance, and reliable design and verification. She had fulfilled positions such as General Chair, Program Chair, for many prestigious IEEE conferences such as IEEE VTS, IEEE ETS, IEEE On-Line Test Symposium. Dr. Anghel has been recipient of several Best Paper and Outstanding Paper Awards and she has published more than 130 publications in international conferences and symposia. She has supervised 24 PhD Students. From 2016 to 2020 Dr. Anghel was Vice President at Grenoble INP, in charge of Industrial relationships and she is currently Scientific Director for Grenoble INP.

Riccardo CANTORO received the MS degree and the PhD in computer engineering from Politecnico di Torino, Italy, in 2013 and 2017, respectively. He is currently a researcher with the Department of Computer Engineering of the same university. His research interests include functional testing of SoCs and memories, data analysis, and machine learning applied to test and diagnosis. He was involved in the program committees and organizing committees of several IEEE conferences and workshops and is currently the Program Co-Chair of the Test Technology Educational Program of the Test Technology Technical Council. He is a member of the IEEE and the IEEE Computer Society.

Riccardo MASANTE received the MS degree in Electronics Engineering - Embedded Systems from Politecnico di Torino, Italy, in 2020. His thesis focused on finding new procedures to write ASM programs able to reach high fault coverages for path delay faults. He received the graduation award "Techniques for aging detection of processors in a safety-critical environment".

Michele PORTOLAN received his PhD in Microelectronics in 2006 from Grenoble-Institute of Technology (Grenoble-INP), France. In 2003 her received both a Masters in Telecommunication Engineering from Grenoble-INP a Master in and Electronics Engineering and Politecnico di Torino, Italy, as part of a Double-Degree program. He has been an IEEE member since 2007 He is currently Associate Professor at Grenoble-INP (since 2013), member of the TIMA Laboratory. Previously he was a Member of Technical Staff at Bell Labs Alcatel-Lucent (now Nokia) in Ireland and France from 2007 to 2013. His main research themes are Digital Testing, Automation, Embedded Systems and Reliability. He is one of the signing member of the IEEE 1687-2014 standard and part of several Working Groups, such as IEEE P1687.1. He is the author of papers in Journals and International Conferences and has been granted several Patents from the USPTO and the EPO.

Sandro SARTONI received the MS degree in Electronics Engineering - Embedded Systems from Politecnico di Torino, Italy, in 2019, where he is currently pursuing his PhD in Computer Engineering. His research focuses on devising new strategies for functional test of path delay faults in advanced semiconductor technologies, also taking into account how such technologies are affected by aging. He is a student member of IEEE and IEEE-HKN, the honor society of IEEE.

Matteo SONZA REORDA received the MSc degree in electronics and the Ph.D. degree in Computer Engineering from Politecnico di Torino, Italy, in 1986 and 1990, respectively. He is currently a Full Professor with the Department of Control and Computer Engineering of the same institution. He published more than 400 papers in the area of test and fault tolerant design of reliable circuits and systems, receiving several Best Paper Awards at major international conferences. He is involved in numerous research projects with companies and other research centers worldwide. He is a Fellow of the IEEE.