A Novel Indirect Methodology based on Execution Traces for Grading Functional Test Programs

A Novel Indirect Methodology based on Execution Traces for Grading Functional Test Programs

Francesco Angione* *IEEE Student Member*, Paolo Bernardi* *IEEE Senior Member*, Andrea Calabrese* *IEEE Member*, Lorenzo Cardone* *IEEE Student Member*, Stefano Quer* *IEEE Senior Member*, Claudia Bertani†, Vincenzo Tancorre†
*Department of Control and Computer engineering, Politecnico di Torino, Turin, Italy
†STMicroelectronics, Agrate Brianza, Italy

***Abstract*—Developing functional test programs for hardware testing is time-consuming and experience-wise. A functional test program's quality is usually assessed only through expensive fault simulation campaigns during early development. This paper presents indirect quality measurements of fault detection capabilities of functional test programs to reduce the total cost of fault simulation in the early development stages. We present a methodology that analyzes the instruction trace generated by running functional test programs on-chip and building its control and dataflow graph. We use the graph to identify potential flaws that affect the program's fault detection capabilities. We present different graph-based techniques to measure the programs' quality indirectly. By exploiting standard debugging formats, we individuate instructions in the source code that affect the graph-based measurements. We perform experiments on an automotive device manufactured by STMicroelectronics, running functional test programs of different natures. Our results show that our metric allows test engineers to develop better functional test programs without basing their development solely on fault simulation campaigns.**

***Index Terms*—Functional Test Programs, Software-Based Self-Test, Burn-In Stress, System-Level Tests, Data Flow Analysis, Hardware Testing, and Fault Simulation.**

I. Introduction

THE silicon manufacturing test flow comprises different phases. On the one hand, structural methods, like Built-in Self-Test (BIST) and other scan-oriented techniques, look appropriate for screening out devices affected by permanent faults. However, they fail to catch all problems related to system module interactions since those techniques transform the device into a non-functional mode and do not adequately protect the device during its lifetime. On the other hand, functional test strategies, such as Software-Based Self Tests (SBSTs), have become very popular to test devices during their lifespan [1], [2], [3], efficiently substituting redundancy-based techniques (e.g., lockstep) for online testing, or for testing at-speed during manufacturing test [4], [5], [6]. Some companies have recently introduced an additional manufacturing test flow phase, called System-Level Test (SLT), to highlight problems escaped from previous test phases and residual marginal behaviors [7], [8], [9], [10] due to system components interactions, computation and communications peripherals, and complex hardware-software interactions. SLT often involves booting an operating system and scheduling applications to mimic the in-field behavior.

Moreover, there are concerns on how to develop a functional test program for online testing of CPU fleet in data centers capable of capturing faults provoking Silent Data Corruptions (SDCs) [11], [12] of different nature, i.e., transient and permanent. SDCs are very hard to detect defects in manufacturing; they require specific voltage, frequency, temperature, workload, and high test iterations.

Functional test program creation requires manual and/or iterative efforts and is graded by using fault simulation tools [13] before functional verification correctness. Unfortunately, the fault simulations are exceptionally time-consuming [14], forcing test engineers to wait for a significant amount of time and, in case of unsatisfactory results, iterate the entire process, even with automated generation engines [15].

Due to their requirements of exercising the device as a whole, combined with long test execution time, SLT applications and SDC-detecting applications share the same Achilles' heel: the extremely high cost often leading to the unfeasibility of fault simulation campaigns, and consequently, the lack of a quality metric for such class of functional test programs.

This paper proposes a novel and indirect methodology to quickly analyze a functional test program's quality regarding fault detection capabilities without requiring fault simulation campaigns in early development stages. The methodology loads the instruction trace of a functional test program directly produced by the chip, controlled by a hardware debugger or an Instruction Set Architecture (ISA) simulator. The instruction trace is converted into a Control and Data Flow graph (CDFG) representation. Then, the data and control flows between registers and memory locations are analyzed. The graph representation allows the definition of high-level graph-based metrics, which represent whether the functional test programs effectively carry their computed value to a diagnostic point, i.e., a software signature in memory or a hardware diagnostic register. The proposed metrics are approximations of lower-level metrics, such as the one representing toggle and fault coverage. Thus, they indicate the quality of functional

Francesco Angione, Paolo Bernardi, Andrea Calabrese, Lorenzo Cardone, Stefano Quer are with the Politecnico di Torino, Torino, Italy. E-mail: <name.surname>@ polito.it.
Claudia Bertani and Vincenzo Tancorre are with STMicroelectronics, Agrate Brianza, Italy. E-mail: <name.surname>@st.com.

test programs in propagating the computed values, exciting the registers' bits, and diversifying data patterns. Low connectivity of the CDFG indicates that some flaws may affect the program (e.g., previously computed results are overwritten or memory locations are not included in the signatures), potentially reducing its ability to detect faulty behaviors. To provide fine-grain feedback to test engineers, the graph analysis is combined with the executable file and the source code to locate code lines affecting the computed metrics. The primary objective of this work is not to replace fault simulation campaigns. On the contrary, the target is to prove that the proposed methodology significantly accelerates the development process for functional test programs in the early development stages. The core idea is to provide propagation information at the assembly instructions level, especially for SBSTs. By leveraging this methodology, the efficiency of generating high-quality test programs can be significantly enhanced, reducing the time and effort required by test engineers.

The proposed methodology is applied to an automotive device manufactured by STMicroelectronics, a medium-size System-on-Chip (SoC) used in safety-critical applications. Experimental results are provided in different contexts. First, the proposed methodology has been used to analyze already developed Software Test Libraries for core online testing [1]. Then, it evaluates system-level test applications based on different versions of an existing Real-Time Operating System (RTOS). Finally, to estimate the boost in the development process, we develop two SBSTs programs from scratch using an Evolutionary Optimizer called μGP [15]. The SBST development time is compared between a flow using the proposed metrics and a flow based on fault simulation results. The experimental results demonstrate that low connectivity of the CDFG always correlates to low fault coverage values. In all cases, the newly proposed metrics have been highly efficient in developing and improving functional routines and avoiding extraordinarily long and repetitive fault simulation campaigns.

It has to be noticed that this work extends the conference paper by Angione et al. [16] in the following directions. We rewrote the core algorithms to evaluate the connectivity metric to make the process faster, less memory-consuming, and more scalable; the new procedure is at least one order of magnitude faster than the original one. We introduced new register-based metrics and made our analysis more accurate, moving from a coarse-grain register-transfer level analysis to a fine-grained bit-level estimation. We improve our analysis with a strategy to perform fault injection in our trace sequences and change the control flow of the test programs. We adopt a reverse-engineering approach to locate software problems directly in the original high-level software application.

The paper is organized as follows. Section II introduces some background on functional test techniques and their evaluations. Section III shows the proposed methodology. Section IV illustrates our experimental results. Finally, Section V summarizes some final considerations and reports some possible future works.

II. Background

This section briefly introduces the main state-of-the-art functional test methodologies and the more strictly related works.

A. Functional test programs

Traditionally, structural test methods are predominant in manufacturing testing, as they are automated and reach high fault coverages. However, alternative techniques for manufacturing tests, such as functional test programs, are emerging and being adopted by companies due to the increasing complexity of modern devices and the test equipment's limitations for storing test patterns. Functional test programs can cover different hardware testing aspects thanks to their flexibility, low intrusiveness, and at-speed execution:

- Software-Based Self Tests (SBSTs) test specific modules during the manufacturing test phase [4] or in mission mode [1]. SBST executes functional test patterns exploiting the processor instruction set on different components [25], [26], [27], [28].
- Test During Burn-In combines the Burn-In phase, i.e., removing the infant mortality of defective devices [29], with functional test/stress programs [5], [6], [20].
- System-Level Test consists of executing a functional application mimicking the device's mission mode [7], [9], [10], [30]. For example, booting and running a Real-time Operating System (RTOS) as a benchmark is a standard option.

All these functional test programs share the same underlying concepts: They exploit CPU instructions to apply data patterns to a target module (or a group of modules) to collect their computed values or states and compare them with the expected output to decide if a fault has been detected. The significant advantages of SBSTs are flexibility, low intrusiveness in system design, low power consumption, and reuse of already present hardware. Those advantages make SBSTs the most desirable approach for online testing of microprocessor-based systems.

B. Related Works

Table I presents a qualitative evaluation of the state-of-the-art assessment methodologies based on different aspects, ranging from the necessity of an experimental setup, its cost in terms of equipment and resources, the abstraction level at which the assessment is carried out, and the target fault model. The last two columns are the most significant as they represent the controllability and observability obtained by the methodology, i.e., the capability of controlling where a fault is inserted and where its effect can be observed.

The beam experiments have been used to access functional test programs in a real-case scenario (neutron beams). However, it is a fast but expensive methodology targeting only transient faults without knowing where the fault is located.

A debugger-based fault injection lowers the abstraction to the assembly level, is capable of injecting transient faults in the user registers, and can observe whether the application

TABLE I: State-of-the-art methodologies for functional test program assessment for hardware testing of CPU-based SoCs.

Method	Experimental Setup	Cost	Abstraction	Execution Time	Fault Model	Controllability	Observability
Beam experiments [17]	Real Device	Very High	Application Level	Low	Transient	Random	Application Error
Debugger-Based Fault Injection [18], [19]	Real Device	High	Assembly Level	Medium	Transient	User Registers	Application Error
Proposed methodology [16]	Real Device or ISA simulator	Medium	Assembly Level	Medium	Transient Permanent	User Registers	Data Propagation
Micro-Architectural Fault Injection [17]	Architectural description	Medium	Micro-Architectural Level	Medium	Transient Permanent	Arch. Registers Modules	Arch. Registers and Application errors
Toggle coverage analysis [20]	Logic Sim.	High	Signal Level	High	Toggle	Signal	Signal
RTL Fault Simulation [21], [22]	Fault Sim.	High	Behavioural level	High	Transient Permanent	RTL Signal	RTL Outputs
Gate-level Fault Simulation [23], [24]	Fault Sim.	Very High	Gate level	Very High	Transient Permanent	Gate-level Signal	Gate-level Outputs

crashes or not. It requires the device, as the beam experiments methodology, and a debugger capable of corrupting register values; thus, even if it has an acceptable execution time, it remains an expensive technique.

Micro-architectural fault injection relies on the architectural description of the SoC, having a medium cost, including the setup of the micro-architectural simulation and description on a medium-performance server. It can inject transient and permanent faults in memory elements at a micro-architectural level in a reasonable execution time. The fault can be controlled in the architectural registers or a specific module, and it is observed as an application error or a mismatch in the architectural registers.

Netlist-based methodologies can rely on different types of fault models, and they are very accurate when controlling the fault location and observing its effect on the outputs. On the one hand, toggle coverage analysis can be used before gate or RTL fault simulation to check if a fault can be controlled in a specific line. Thus, controllability and observability are bound to the signal level. Meanwhile, RTL fault simulation can inject the fault in the RTL signal and observe the RTL outputs to understand if the fault is excited and propagated to the outputs; gate-level fault simulation can inject the fault on the gate-level signals and observe if it can be propagated the gate-level outputs.

The proposed methodology analyzes how data propagates among instructions to observable points, identifying blocking instruction overriding computed results and potentially affecting the final fault coverage. This approach can be added in the early phases of functional test program developments to guide test engineers, avoid repetitive fault simulation campaigns, and, at the same time, provide fine-grained feedback on the developed functional test program. The methodology shares similarities with the micro-architectural fault injection methodology. However, a micro-architectural description of the SoC is not always available, and it has a cost to configure it properly. On the contrary, the proposed methodology relies on analyzing the instruction trace that can be extracted from the actual device, an Instruction Set Architecture (ISA) simulator, or a logic simulation (often integrated into test benches capable of executing functional programs). This approach makes the proposed methodology cost-effective and viable for many testing scenarios.

In terms of evaluation, the proposed methodology exhibits parallels to both debugger-based and microarchitectural fault injection techniques. However, debugger-based fault injection evaluates the resilience of the functional test program to perturbations in user register values throughout its execution. While it is technically possible to check the system state after every instruction, doing so incurs a high computational cost due to the need to compare against a golden model during functional testing. In contrast, micro-architectural fault injection provides control and observability over all micro-architectural registers and the application output. However, it does not offer feedback on which parts of the functional test program are essential for capturing faulty behavior. Like debugger-based fault injection, it grades the test program over its entire execution without pinpointing specific areas for improvement. The proposed methodology, on the other hand, grades functional test programs from a test program fault detection perspective by unrolling and analyzing their execution. It evaluates the quality of the test program by relying solely on the user's view of the registers and the punctual data propagation among them for each instruction. This approach highlights whether a fault may be masked by an instruction or propagated toward the end of the test program, providing valuable feedback at the instruction level. This fine-grained feedback makes the proposed methodology particularly useful for improving the quality of functional test programs.

III. Proposed Methodology

A fault simulation campaign reports only the fault coverage obtained without indicating the flaws that affected the functional test program. The absence of analysis tools fuels the disappointment in identifying functional test program weaknesses that may prevent reaching a very high fault coverage. In other cases, fault simulation campaigns are not affordable for SLT applications and SDC-oriented functional test programs. The proposed methodology is based on the architectural observations reported in the following example.

Example 1: Consider the assembly code segment reported in Table II. The table includes three instructions for the *RiscV* instruction set architecture and illustrates their operation.

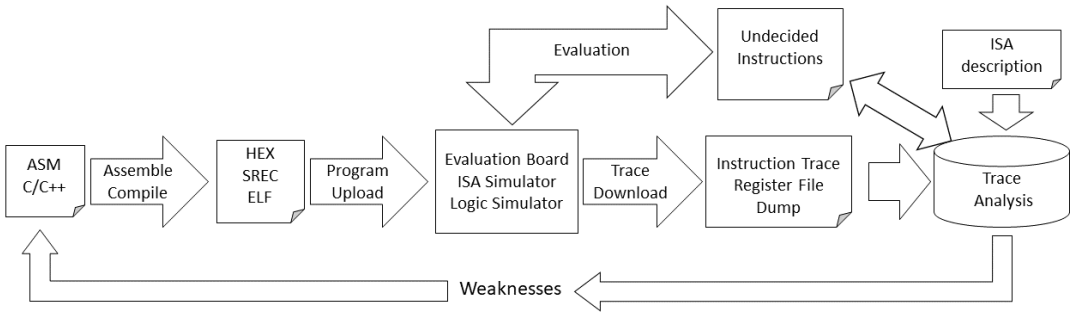


Fig. 1: Workflow for the proposed methodology.

TABLE II: An example of *RiscV* code segment with the operation performed.

Cycle	Instruction	Operation
1	addi x19, x18, 192	x19 = x18 + 192
2	sub x6,x6 x18	x6 = x6 - x18
3	addi x6, x19, 35	x6 = x19 + 35

The data written in x6 by instruction number two is immediately overwritten by instruction number three. This behavior is a *Write-After-Write (WAW)* hazard in computer architectures. Even if there are cases where a WAW behavior may have a specific meaning, they somehow correspond to undesirable situations with information flow disruptions. In principle, these situations should never appear in ASM or compiled C/C++ functional programs and should be prevented by the compiler.

Vice-versa, the reading operation of instruction three preserves the write operation on register x19 performed by instruction one; this behavior is named *Read-After-Write (RAW)* hazard. RAW behaviors indicate standard operation flows and are beneficial.

Overall, instruction one positively affects the fault coverage if the destination registers are propagated to a software signature in memory during fault simulation, i.e., faults in the ALU potentially propagate to the signature. On the contrary, instruction two is not beneficial to fault coverage as the value of x6 is overwritten and cannot be propagated. Consequently, instruction two can be removed, or the program needs to be modified to unlock a possible propagation of x6.

An analytical evaluation of the number of potential problems, such as the one shown in the previous example, could indicate potential flaws in the functional test program from a fault detection capabilities perspective.

From the computer security domain, "dynamic taint analysis" runs a program and observes which computations are affected by predefined taint sources, such as the user inputs [31], [32], [33], [34]. Following the dynamic taint analysis paradigm, the proposed methodology evaluates a functional test program as illustrated in Figure 1.

As represented by the leftmost block of the chain in Figure 1, the input of our methodology is a functional test program written in a high-level language such as C or at low-level assembly code. Compiled programs can be uploaded on an evaluation board, an ISA simulator (or even a logic simulator), which must provide capabilities of dumping:

- The list of executed instructions, with all selected branches and all loops properly unrolled (the instruction trace file).
- All registers that are used during the execution of the machine code (the register log for each instruction within the instruction trace file).

We parse the instruction trace file and transform it into a Control and Data Flow Graph (CDFG) with the proposed trace analysis methodology (represented in the rightmost block of Figure 1). In this CDFG, a vertex represents an instruction, and each direct edge defines the information flow between two instructions.

The Instruction Set Architecture (ISA) (i.e., the top-right block of Figure 1) is used to provide a specific description of registers and instructions, achieving portability among different ISAs. Instructions are categorized by type (e.g., arithmetic, branch, load, store, etc.), with attributes like byte size, destination positions, operand positions, and memory access information. Registers are divided into general-purpose (e.g., from R0 to R31) and special-purpose, with details including type, permissions (read/write), width, aliases, sub-registers, and initial values. This format provides a detailed schema for comprehensively describing a processor's architecture, covering its instruction set and register file. We generate it starting from the ISA documentation.

To build the CDFG, we concentrate on writing and reading operations and their temporal dependency. Thus, given a data value d in the trace (either a register or a memory location), we logically introduce two types of edges:

- *WAW* edges representing two write operations on d, with no reading instructions in between.
- *RAW* edges representing a write operation followed directly by a read operation on d.

WAW sequences occurring during the instruction flow over a shared addressable location cause a value to be overwritten, most likely leading to a loss of fault coverage. Conversely, RAW sequences propagate values along with the execution flow, and they can reach an observation point, possibly leading to an increase in fault coverage.

Once the CDFG has been built, we analyze it to extract graph-based metrics based on the *Connectivity* metric proposed in [16]. The connectivity identifies the program's quality at propagating the computed values, exciting the registers' bits, and diversifying the data patterns. To provide fine-grain feedback to test engineers, the proposed methodology provides feedback pinpointing the lines in the source code. Moreover, some cases exist where instructions (mainly control flow instructions) cannot be labeled. Consequently, the proposed methodology permits re-evaluating them by re-executing the

functional program and corrupting the registers used by the instructions to evaluate if they affect the control flow of the program [18]. This information can be combined to grade the quality of the functional procedures, rectify their problems, and insert proper signature checks to improve their fault detection capabilities.

A. The Connectivity Algorithms

In this section, we show how to classify each instruction as black (blocked) or green (not blocked), depending on whether it propagates or does not propagate the computed results towards the end of the execution.

The simplest version of the algorithm is described in [16]. It proceeds in three steps: It first creates WAW and RAW edges, essentially building the graph; then, it visits the graph a first time, following WAW edges and coloring nodes as red or green; finally, it visits the graph a second time, following RAW edges, and coloring nodes as green or black. The following example illustrates the overall flow of this procedure.

Example 2: Table III reports a short snapshot of a generic RiscV-like instruction trace file from a test routine for the CPU adder unit. The first column indicates the order of execution; the second and third columns report the address and the mnemonic code of the relative instruction; columns SRC and DST specify the sources and destinations of each instruction.

TABLE III: Instruction sequence, source, and destination operands.

Cycle	Address	Instruction	SRC	DST
1	0x0000_0002	c.sub r19, r6	r19, r6	r19
2	0x0000_0004	addi r6, r19, 35	r19	r6
3	0x0000_0008	addi r6, r19, 192	r19	r6
4	0x0000_000C	addi r6, r6, r1	r6, r1	r6
5	0x0000_000E	c.sub r19, r6	r19, r6	r19
6	0x0000_0010	c.sub r6, r3	r6, r3	r6
7	0x0000_0014	addi r6, r19, 35	r19	r6

Once we have collected this information, the most straightforward approach runs through the steps illustrated in Figure 2:

1) In the first step, represented by Figure 2a, we build a CDFG in which each instruction is mapped onto a vertex, and the data flow is represented through edges. We use two types of edges: RAW (reported on the left-hand side of the graph) and WAW (reported on the right-hand side).
2) During the second stage, Figure 2b, we perform a visit of the CDFG following WAW edges. Vertices are colored either red or green.
3) During the third and last step, Figure 2c, we perform a visit of the CDFG following RAW edges. Red vertices may become green or black.

The previous process can be optimized by realizing that WAW arcs are unnecessary to obtain the final result. Practically speaking, we perform the WAW visit while building the CDFG and color each instruction in a single backward visit of the CDFG. This process reduces the overall computation costs and improves memory efficiency. Algorithm 1 presents our graph visit, starting from the final instruction and proceeding backward on the edges [35], [36].

In line 1, the algorithm creates an empty set representing the values read along the program. The set must have the property of having only a single instance for each value. In lines 2-7, it stores in this set all observable registers defined

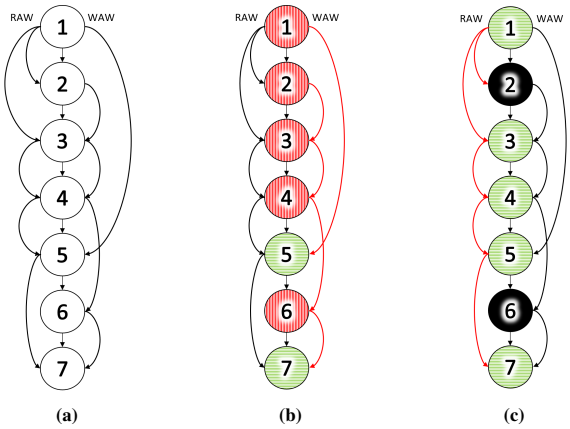


Fig. 2: The CDFG of the code snippet of Table III is represented in Figure (a). Figure (b) illustrates the vertex-coloring process obtained following WAW edges. Figure (c) shows the colors obtained at the end of the RAW visit.

Backward_Visit (G)
1: read_set = $\{\}$
2: **foreach** (register in ObservableRegs) **do**
3: read_set.insert(register)
4: **end for**
5: **foreach** (address in ObservableMemoryAddresses) **do**
6: read_set.insert(address)
7: **end for**
8: **foreach** (node in G in reverse) **do**
9: $node_{color} = BLACK$
10: **foreach** (destination in $node_{dst}$) **do**
11: **if** (read_set.contains(destination)) **then**
12: read_set.remove(destination)
13: $node_{color} = GREEN$
14: **end if**
15: **end for**
16: **if** ($node_{color} == GREEN$) **then**
17: **foreach** (source in $node_{src}$) **do**
18: read_set.insert(source)
19: **end for**
20: **end if**
21: **end for**

Algorithm 1: A one-step algorithm colors all nodes in a single backward visit of the CDFG of the instruction trace.

by the user and the memory locations in which the system (or the operating system) can verify if the functional test program ends correctly (i.e., all memory location storing the signature or hardware peripherals capable of triggering an exception depending on the written value). From line 8, the procedure traverses the graph backward, and in line 9, initializes each node with a black color. When a node has an operand among the destinations read by the observable read set (a memory region or a register) or another operation, it assigns the green color to that node; then, it removes the node from the set of locations that have been read. Since such a register is no longer in the list of read registers, when a new operation tries to write a value in that position, we know one of the following operations will overwrite it. In lines 17-19, if the information contained in the destination registers is propagated

to the observable points, we add the source registers of the current instruction to the list of read values.

Algorithm 1 is significantly more efficient than the original ones proposed by Angione et al. [16] regarding memory occupation and computation time. The new algorithm does not need to create the graph edges, thanks to the backward visit: At each step it already knows whether the current information is propagated to the end of the program or it is overwritten, without the necessity to follow the information flow in the forward direction like the original procedure. For load and store instructions, memory addresses are computed using the actual values of the used registers. Memory addresses are manipulated as virtual registers; thus, they are introduced in the destination for every node in Algorithm 1. The instruction trace can also include instructions with multiple destinations (registers or memory addresses) as the new algorithm automatically manipulates a destination list for each node (i.e., instruction). We consider data values as propagated if a node has at least one green destination, i.e., the instruction propagates at least some partial result. To illustrate this process, the following example presents a code snippet with load, store, and arithmetic instructions.

Example 3: The code snippet reported in Table IV (and derived from the PowerPC VLE ISA) includes load and store instructions from and to memory locations in addition to arithmetic operations.

TABLE IV: Instruction sequence, source, and destination operands.

Cycle	Instruction	SRC	DST
1	e_stb r0, 0(r1)	r0, r1	mem(0+r1)
2	slli r2, r0, 35	r0	r2
3	e_lbz r2, 0(r1)	mem(0+r1), r1	r2
4	e_add2i r2, r1, r2	r1, r2	r2
5	e_stb r2, 0(r1)	r2, r1	mem(0+r1)
6	subfze r1, r3	r1, r3	r1
7	e_add16i r1, r0, 35	r0	r1

Figure 3 includes the coloring scheme returned by Algorithm 1. For example, the first instruction stores a value in the location pointed by register r1. This location is referenced as a source operand by the third instruction. Therefore, instruction one is finally labeled as green (light-gray). Moreover, instruction five is marked as green because the value written in memory is propagated till the last instruction.

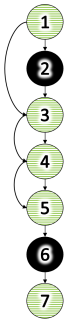


Fig. 3: An example in which our algorithm is applied to a code section includes arithmetic operations and load/store instructions.

During the process represented by Algorithm 1, conditional branches are left undecided and temporarily colored in orange. After that, we visit the CDFG to find the alternative branch among the instructions following the conditional statement. If the branch's alternative address cannot be found, the branch is colored in black. If the branch alternative can be reached, the visit checks whether the incorrect branch execution leads to a different data flow and control flow and colors the branch in black or green. The following two examples illustrate this procedure.

Example 4: The code reported in Table V (PowerPC VLE ISA) includes a branch (line 3) generated by a simple if-then-else construct (line 2).

TABLE V: Instruction sequence, source, and destination operands.

Cycle	Address	Instruction	SRC	DST
1	0x0000_0000	e_li r0, 0		r0
2	0x0000_0004	cmpl r0, 1	r0	cr
3	0x0000_0008	e_beq jump	cr	
4	0x0000_000C	cmpl r0, 0	r0	cr
5	0x0000_0010	e_add16i r1, r2, r3	r2, r3	r1
6	0x0000_0014	jump: e_add16i r1, r0, r1	r0, r1	r1

Figure 4a reports the CDFG of this code snippet. RAW edges, not explicitly computed by Algorithm 1, are reported to highlight the information flow. Branch nodes are initially left as undecided. Then, a second graph visit evaluates the instruction flow under the hypothesis of taking the wrong branch path. The result is represented in Figure 4b. In this case, the instructions that would be reached if the branch decision was wrong are included in the instructions that follow the branch itself. A new orange-colored edge is added to the figure to highlight this situation, and it appears that some instructions would not be executed if the branch was wrongly executed (taken if it was not taken, and vice-versa).

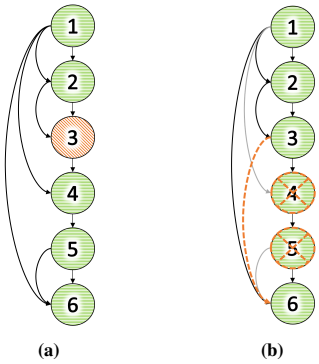


Fig. 4: The graph example includes a branch instruction.

The instructions that may be skipped are green. Therefore, the value of the final registers may change, compromising the execution. Indeed, the branch node is labeled green. Conversely, if all skipped instructions had been labeled black nodes, they would not have contributed to the data flow. The branch would be black.

Example 5: Let us focus on the code reported in Table VI (PowerPC VLE ISA). Following Example 4, we initially leave the branch node undecided. Then we determine that the instruction flow would not change under the hypothesis of taking the wrong branch path, i.e., all skipped instructions are labeled in black. As a consequence, the branch is colored in black. The outcomes of the two visits are illustrated in Figure 5b.

This approach colors branches (in if-the-else statements) only when both paths can be found in the instruction trace. Unfortunately, there are cases in which one of the paths is not present in the instruction trace. One possible solution to this problem is to adopt a guided fault injector, performing fault injections in undecided branches to force a change in the execution trace. Figure 6 graphically represents the action

TABLE VI: Instruction sequence, source, and destination operands.

Cycle	Address	Instruction	SRC	DST
1	0x0000_0000	e_li r0, 0		r0
2	0x0000_0004	cmpl r0, 1	r0	cr
3	0x0000_0008	e_beq jump	cr	
4	0x0000_000C	e_add16i r1, r2, 1	r2	r1
5	0x0000_0010	e_add16i r1, r0, 1	r0	r1
6	0x0000_0014	jump: e_add16i r1, r0, r2	r0, r2	r1

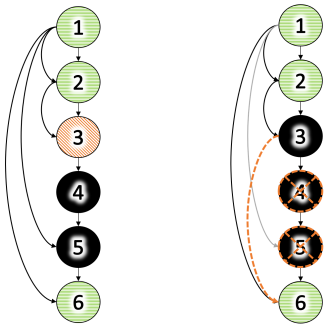


Fig. 5: Another branch instructions example.

performed by the fault injector. We consider the golden instruction trace on the left-hand side, identifying the undecided branches (in instruction 1). Using a fine-grained fault injector, we corrupt the condition where the branch is taken, forcing the execution of the other branch. Exploring both branches, we effectively visit the whole execution space; thus, we can gather precise information about undecided branches of the original instruction trace, i.e., if it changes the CDFG.

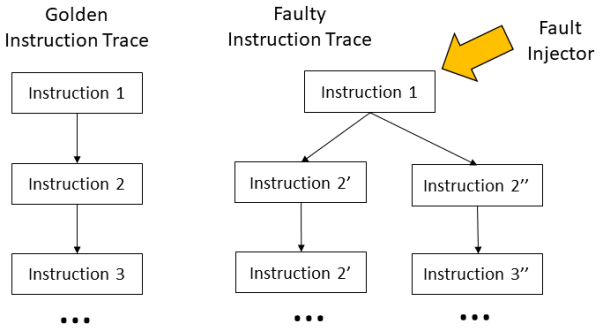


Fig. 6: Fault injection in branch-control registers for undecided branches to improve the representativity of the execution trace,

B. Graph-based Metrics: The Connectivity Metrics

The CDFG built by the coloring algorithms permits the extraction of graph-based measurements. The idea is to use these measures as indirect quality indicators of the functional test program. The quality is strictly related to the fault detection capabilities of the code and helps test engineers boost the program development process. We define three metrics to indicate the program's quality from various aspects.

The *Connectivity* (C) metric, as proposed in [16], for a generic program can be defined as follows:

$$C = \frac{\sum_{i}^{GreenInstr} \left(\frac{1}{Instr_i Dests} \sum_{j}^{Instr_i Dests} GreenDests \right)}{TotInstructions} \quad (1)$$

The formula computes the percentage of instructions that propagate data values toward the end of the functional test program (i.e., possibly to an observable point, such as a software signature in memory or a diagnostic register). It essentially proceeds in two steps. As instructions may have multiple destinations in the computation between parenthesis, a connectivity value is assigned to every CDFG node. After that, it evaluates the sum of connectivity values of all green instructions (weighted by multiple destinations), and it divides this contribution by the total number of instructions.

The previous metric does not consider the number of bits that toggle in the destination. For example, if a simple *add* instruction on a 32-bit register is executed between R1 and R0, using as destination R2, with R1 equal to 1 and R0 equal to 0, only one bit is changed in the destination register. Different faults can be excited and observed as different data patterns may activate different circuit logic cones. Therefore, the previous metric can be improved by computing the number of bit flips. As a consequence, we defined a new metric called *Register Bit-Level Connectivity* (C_{RBL}). Each instruction propagating its data value (i.e., the basic connectivity metric) represents the number of bits that toggle (i.e., change from 0 to 1 or vice versa). The Register Bit-Level Connectivity can be computed as illustrated by the following equation:

$$C_{RBL} = \frac{1}{N_{regs}} \sum_{i=0}^{N_{regs}} \frac{\sum_{j=0}^{R_{i_{bits}}} \begin{cases} 0 & \text{if } R_{t_{i,j}} = 0 \\ 0.5 & \text{if } R_{t_{i,j}} = 1 \\ 1 & \text{if } R_{t_{i,j}} > 1 \end{cases}}{R_{i_{bits}}} \quad (2)$$

Where $R_{t_{i,j}}$ represents the number of total transitions during the whole execution of bit j of the i^{th} register and $R_{i_{bits}}$ represents the total bits of register i^{th}. In this way, the formula provides information about registers that undergo two transitions (from 0 to 1 and vice versa), independently from the length of the test program. Notice that we consider each instruction a black box, as we know only its input and output states. This choice allows the generalization of the metric and relies on the correctness of the instruction trace and the register file. A low value of register bit-level connectivity means that some transitions are not exercised in the registers and that we potentially use bad test data patterns for capturing faulty behavior.

To combine the number of toggles per bit (dependent on the length of the test program) that propagate their value, we introduce the *Register Average Bit-Toggle Connectivity* (C_{RBT}):

$$C_{RBT} = \frac{\sum_{i=0}^{N_{regs}} \sum_{j=0}^{R_{i_{bits}}} R_{t_{i,j}}}{\sum_{i=0}^{N_{regs}} R_{i_{bits}}} \quad (3)$$

Where $R_{t_{i,j}}$ and $R_{i_{bits}}$ have the same meaning introduced for Equation 2. The formula computes the arithmetic mean

of the number of toggles among the total register bits. Notice that adding small values or values with many zeroes, using division instructions with a large dividend, and multiplying large numbers together produce uniform outputs, i.e., similar to one of the input operands. These values can produce significantly high C_{RBT} values even when most bits remain unchanged.

The average mean computed in the previous formula can be easily substituted with a geometric mean to evaluate programs with unbalanced toggling activity more appropriately. Thus, we can define an additional metric based on the geometric mean of the toggle values of test programs. We call it *Register Bit-Toggle Tendency Connectivity* (C_{RT}), as the following equation shows:

$$C_{RT} = \exp\left(\frac{1}{\sum_{i}^{N_{Regs}} R_{i_{bits}}} \sum_{i}^{N_{Regs}} \sum_{j=1}^{n} \ln R_{t_{i,j}}\right) \quad (4)$$

Where $R_{t_{i,j}}$ and $R_{i_{bits}}$ have the same meaning introduced for Equation 2. The *Register Bit-Toggle Tendency Connectivity* measures the number of average bit transitions for instructions that propagated their computed values. It is equal to zero when there are no transitions on some bits of registers.

To summarize, the presented metrics allow us to grade the quality of a functional test program in the following directions:

- The *Connectivity metric* represents the flow of data values among the instructions [16].
- The *Register Bit-Level Connectivity* evaluates the flow of bit transitions among registers.
- The *Register Average Bit-Toggle Connectivity* computes the arithmetic average of bit transition among registers.
- The *Register Bit-Toggle Tendency Connectivity* measures the general tendency of bit transitions for data patterns.

C. Weaknesses identification in the source code

For increasing productivity during the development of functional test programs, it is crucial to highlight potential flaws or weaknesses in the source code to prevent fault masking by WAW edges, a scenario that can quickly go unnoticed but has significant implications regarding fault coverage, Figure 7 shows our weaknesses identification flow.

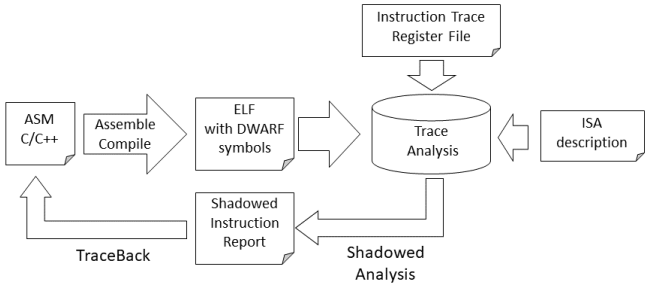


Fig. 7: Weaknesses identification flow.

The source code is usually written in assembly or a high-level programming language (such as C or C++) in multiple files linked together by the compiler in the executable. The proposed methodology analyzes the instruction traces and the register files incremented with information from the executable file (in ELF format) and the debugging symbols (in DWARF format) produced by the compiler. The methodology identifies black instructions that fail to propagate their computed data and creates a report highlighting them. A traceback step allows us to link the identified weaknesses to the source code. The decision on rectifying the problem is left to the test engineer, who can either remove the instructions or add a partial signature computation. Moreover, a black instruction can potentially mask (or shadow) the results of other instructions, further reducing the proposed metrics and impacting the fault coverage. The following subsections briefly describe the traceback and shadowed instruction analysis.

1) Traceback From Assembly to Source Code: The reconstruction of the source code from assembly instruction traces is a very well-studied topic in reverse engineering literature [37], [38]. Therefore, the proposed methodology does not provide any innovative methods. However, it relies on providing feedback by tracing the identified weaknesses in the source code, adopting the commonly used dwarf library (`libdwarf`). This library can access the DWARF debugging information and symbols in executable and object files. The compiler must produce the executable file (in ELF format) with the DWARF debugging information and symbols to allow the proposed methodology to trace the identified weaknesses to the source code. The debugging information contains all the data types, sections, call frame information, and the line numbers in the source code [39]. We use the table storing the line numbers to identify in the source code the weaknesses discovered by the proposed methodology in the assembly code.

2) Shadowed Instruction Analysis: The shadowed Instruction analysis identifies the black node (i.e., instructions) overwriting data destinations and disrupting the data flow of other instructions. It parses the CDFG in reverse order to compute how many instructions a given instruction blocks by reconstructing the disrupted data flow from the sink node (the instruction where the dataflow is disrupted) up to the beginning of the CDFG. Thanks to this analysis, we can find instructions that overshadow many others, producing a cascade effect and turning multiple dependent instructions green with a single change to the code.

IV. Experimental Results

A. Experimental setup

The proposed methodology is applied to an industrial, automotive SoC manufactured by STMicroelectronics with 40 nm technology. This SoC features multiple cores (PowerPC VLE ISA) and many peripherals such as DMA, timers, and several communication modules. It is usually employed in safety-critical applications in modern vehicles. Overall, the unit includes about 20 million gates, and each one of its CPUs includes about 1.5 million of stuck-at faults. Figure 8 illustrates the experimental setup used to validate the methodology. The functional test programs are loaded using a custom debugger on a Xilinx ZCU 104 [40]; afterward, the instruction trace is dumped from the on-chip execution and analyzed.

For the sake of comparison, functional test programs are also evaluated on the gate-level netlist description of the

TABLE VII: Functional test programs evaluation for benchmark applications: STL for online testing and RTOSs for SLT.

Program name		Code size [bytes]	Execution Time [cc]	Executed instructions	Connectivity [%]	C_{RBL} [%]	C_{RBT}	C_{RT}	Grading Time [s]	SAF	SAF cov. [%]	Fault simulation CPU time [h]
Adder		2,708	3,388	1,082	95.67	97.66	7.32	6.25	40	20k	92.56	24.39
Multiplier	original	5,612	7,002	1,332	82.20	89.26	12.17	7.50	33	64k	92.30	46.09
	improved	5,584	6,981	1,320	82.65	91.21	12.27	7.80	33	64k	92.30	43.14
Floating Point		13,948	78,286	17,645	97.53	75.78	183.28	8.90	680	70k	90.78	1,407.11
Shifter	original	6,344	7,226	2,079	96.28	87.66	16.81	6.55	80	17k	86.19	169.61
	improved	6,344	7,226	2,078	96.54	89.22	16.92	6.84	80	17k	89.48	39.86
Count-zeros		1,408	3,823	1,112	94.30	80.47	4.92	2.61	43	3k	86.81	12.19
Bit-wise Logical		680	513	146	100.00	90.62	6.73	4.65	5	3k	95.00	5.65
Load-Store		3,016	4,228	1,052	49.81	61.72	11.72	3.86	40	13k	52.54	11.19
BTB		4,476	31,135	3,794	45.41	63.44	4.97	2.73	133	20k	71.16	66.86
RTOS V1	original	130,024	210,816	16,715	24.78	12.94	1.28	1.19	1,108.43	1.5M	NA	7,154.92 (est.)
	enhanced	131,906	467,840	19,895	35.62	15.43	2.29	1.30	1,349.97	1.5M	NA	10,655.26 (est.)
RTOS V2	original	162,795	294,134	37,094	14.41	15.28	1.27	1.25	1,430.82	1.5M	NA	9,982.64 (est.)
	enhanced	164,850	495,858	40,094	20.31	15.97	1.89	1.31	1,546.50	1.5M	NA	11,293.39 (est.)
RTOS V2 enhanced Opt.	Loop opt.	164,936	537,391	40,201	20.29	16.02	1.90	1.31	1,550.67	1.5M	NA	12,239.33 (est.)
	L/S opt.	164,872	535,961	40,094	20.31	16.11	1.91	1.31	1,546.54	1.5M	NA	12,205.75 (est.)
	Loop unroll.	170,168	408,525	30,487	20.91	17.38	0.78	1.22	1,175.93	1.5M	NA	9,281.56 (est.)
	Regs Live	166,028	534,611	39,993	20.39	14.04	0.84	1.21	1,542.64	1.5M	NA	12,176.00 (est.)
	Min dist L/S	164,854	534,948	40,094	20.30	15.97	1.89	1.31	1,546.50	1.5M	NA	12,206.45 (est.)
	all opt.	170,348	439,072	32,849	13.24	14.75	0.59	1.16	1,266.96	1.5M	NA	10,000.07 (est.)

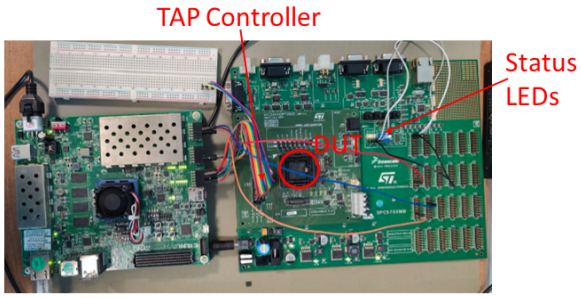


Fig. 8: Experimental setup with FPGA-based debugger.

Device Under Test (DUT) for the stuck-at-fault model using Synopsys's commercial fault simulator *Z01X* and accepting extremely long FSIM time. When these times are excessively long, CPU time estimations are provided only.

B. Evaluation of Functional Test Programs

During the first set of experiments, we analyze SBST programs belonging to a Core Self Test library written in assembly code (avoiding compiler optimizations) [1], [28] developed in about 6 months. SBST programs are used during online testing to target all potential stuck-at-faults (SAF) in CPU modules.

The first section (the one on top) of Table VII reports our findings in this area. To understand the complexity of each program, we include their size (in bytes), execution time (in clock cycles), and the number of instructions executed. Columns "Connectivity", "C_{RBL} ", "C_{RBT}", and "C_{Rt}" indicate the connectivity metrics previously defined, i.e., the *Connectivity metric*, the *Register Bit-Level Connectivity*, the *Register Average Bit-Toggle Connectivity*, and the *Register Bit-Toggle Tendency Connectivity*. Column "Grading Time" reports the execution time. This is mainly due to the extraction of the instruction trace. The last three columns are dedicated to fault-related information and report the number of stuck-at faults (column "SAF") for the module under test, the fault coverage (column "SAF cov."), and the single-threaded fault simulation time for each program.

Although each program has its intrinsic characteristics, the grading time is proportional to the number of executed assembly instructions. Each assembly instruction's structural complexity has a significantly lower influence on the grading time thanks to a clever data structure containing all the necessary information.

The shadowed analysis for identifying weaknesses in the source code allowed us to improve some of the test programs. In particular, we identified a lack of connectivity for the *Shifter* test program due to a single instruction affected by a WAW hazard. Once this issue was rectified, the *Connectivity metric* increased from 96.28% (line "original") to 96.54% (line "improved"). Similar improvements can be seen on the other connectivity-related metrics, and its fault coverage increased by more than 3%, from 86.19% to 89.48%. For the *Multiplier* test program, a few instructions did not propagate their output value to an observable point and were thus marked as useless. In our test case, this observable point was a software signature in memory. Therefore, the useless instructions have been removed from the specific Multiplier SBST. In this way, the execution time and the memory footprint of the program were reduced (ca. 0.5% for ROM footprint and ca. 0.3% for the execution time) while the fault coverage remained 92.3%.

Notice that the above improvements have been obtained with small computational efforts: in all the cases, the time required by the proposed methodology is in the order of seconds, whereas fault simulation requires hours of CPU time. The proposed connectivity metrics provide useful insights about the data patterns used by the functional test programs depending on the nature of the unit under test. For a pipeline sequential module such as the floating point unit, to effectively test the module, the characteristics of the data patterns are high values of registers bit toggle and tendency mean, repetitively applied to allow the sequential logic cones to propagate faults to the outputs. For arithmetic combinational modules like the adder, multiplier, and shifter, efficient test data patterns require only a few transitions among registers to achieve an acceptable level of fault coverage. Regarding combinational input-constrained modules, such as the load and store unit, count-zeros, and bit-wise logical unit, they do not accept the

whole representable range of values. On the contrary, they must use specific test data patterns leading to low register-oriented connectivity metrics, but they must be very effective. Therefore, depending on the nature of the unit under test for SBSTs, register-oriented connectivity metrics can progress in different acceptable ranges for high-quality functional test programs.

The second section of Table VII (the one marked as "RTOS") reports an analysis of an RTOS used for SLT and written in a high-level programming language. The device can run a basic and an advanced version of the Micrium-C Operating System III [41] and different application tasks mainly related to mathematical operations. The RTOS code has been compiled using the optimization level 2 as baseline ("RTOS V1/V2" in the second section of Table VII), and with different optimization strategies that affect the generated machine code in the executable, e.g., loop optimizations, load/store optimizations, loop unrolling, control the register live range, control the distance between load and store of a variable, respectively "Loop Opt.", "L/S Opt.", "Loop Unroll.", "Regs Live", "Min dist L/S" and all the previous optimization in "RTOS V2 enhanced Opt." in the second section of Table VII. Our methodology is motivated by the fact that the fault simulation of these applications and, more generally, of complex system-level test programs is practically unfeasible [9], due to the enormous computation times, estimated to be at least equal to one year.

As the table shows, the RTOS's first version (V1), with a small set of application tasks, has very low connectivity. When this version of the RTOS is enhanced with a signature computation inserted in every context switching between tasks, the basic connectivity increases by roughly 10%. With longer application tasks, version V2 of the RTOS dramatically pushes down the overall basic connectivity metric while slightly increasing the register-oriented connectivity metrics. Enhancing the RTOS with a signature computation in every context switch increases our connectivity metrics.

Compiler optimizations are crucial in determining the number of weak instructions and their positions in the execution trace when grading functional test programs written in high-level languages. Unlike SBSTs (test routines written in assembly code), where code can be precisely modified, high-level code cannot be adjusted as effectively. As expected, experimenting with different optimization strategies (e.g., "RTOS V2 enhanced Opt." in the second section of Table VII) leads to changes in connectivity metrics, which follow the variations in the generated assembly code in different directions. For instance, compared to the baseline "RTOS V2 enhanced," compiled with the optimization level O2 (as shown in the second section of Table VII), introducing different optimization techniques can reduce the code size and/or the number of executed instructions. The latter directly impacts connectivity-related metrics. On one hand, reducing the distance between the load and store of a variable and optimizing loops and load/store instructions can slightly affect these metrics. On the other hand, optimization strategies that increase code size and the number of executed instructions tend to produce more uniform machine code with higher data flow among instructions, thereby increasing connectivity metrics. However, it is important to note that excessive optimization may reduce connectivity-related metrics, as represented by the "RTOS V2 enhanced Opt. - all opt." row. Additionally, it is critical to observe that the generated assembly instructions strongly depend on the structure of the high-level code.

Generally, a high-quality SLT application should exercise the DUT and effectively collect computed results with various data patterns. Consequently, by applying our methodology to different versions of the RTOS, it is easy to understand that the RTOS may fail to capture faults at the system level. Since they strongly rely on control verification points, the BTB and RTOS programs include different undecided branches (i.e., if-then-else statements). As previously discussed, undecided branches without the alternative flow in the instruction trace are marked as undecided. To correctly evaluate them, the functional test programs must be executed twice with the fault injector active to force a change in the control and data flow of the program, using the methodology proposed in [18]. The last column of Table VIII presents the connectivity results after the fault injection in undecided branches. The table shows that the RTOS and the BTB can effectively capture changes in the control flow as the connectivity metric increases after the fault injection. Notice that all metrics evolve like the basic connectivity; however, they are not reported in the table for space.

TABLE VIII: Connectivity metrics post-fault injection for undecided branches (the average injection time is about 2.0 s).

Program name		Connectivity [%]	# Undecided branches	Connectivity increase [%]
BTB		45.41	676	62.54
RTOS V1	original	24.78	1,052	31.53
	enhanced	35.62	1,215	39.66
RTOS V2	original	14.41	9,481	18.64
	enhanced	20.31	9,481	23.77
RTOS V2 enhanced	Loop opt.	20.29	9,496	22.26
	L/S opt.	20.31	9,481	22.25
	Loop unroll.	20.91	2,242	23.09
	Regs Live	20.39	9,428	22.24
	Min dist L/S	20.30	9,481	22.24
	all opt.	13.24	1,192	15.37

C. Developing New and Effective Functional Test Programs

To further prove the effectiveness of the proposed methodology for developing functional test programs, we used an Evolutionary Optimizer μGP [15] for generating them. We adopt the same experimental setup introduced in the previous section to automatically extract the instruction traces and register files for the proposed methodology. Moreover, we use our metrics as fitness values for the evolutionary optimizer μGP.

The targeted fault coverages for generating functional test programs from scratch for both the multiplier and integer divider units were deliberately chosen to meet or exceed the fault coverages achieved by previously developed test programs [1]. These earlier test programs were created through a combination of multiple evolutionary execution and hand-written methods over a development period of approximately six months. The primary objective of this section is to present how the proposed methodology significantly accelerates the

development process for functional test programs while maintaining or surpassing the fault coverage benchmarks of prior approaches. By leveraging this methodology, the efficiency of generating high-quality test programs can be significantly enhanced, reducing the time and effort required.

Figure 9 shows the evolution of a test program for a multiplier unit in terms of connectivity and fault coverage.

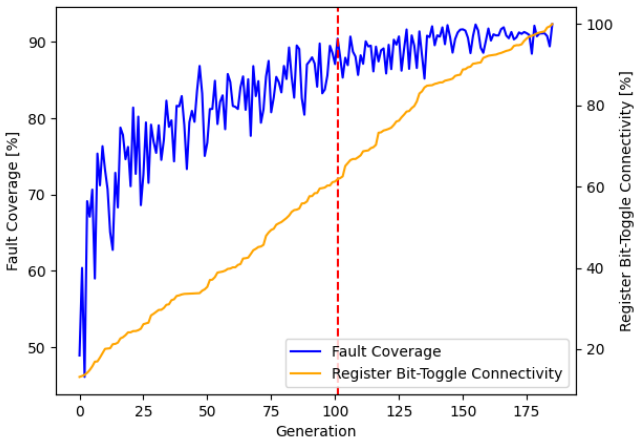


Fig. 9: Evolutionary generation of SBST for the multiplier unit.

The plot illustrates the characteristics of the best programs in each population during its evolution generation after generation (along the x-axis). Due to the structure of the generated test programs, the values of the connectivity metrics are always extremely high; however, they are not reported for the sake of readability. On the contrary, the register bit-toggle connectivity grows even when the other connectivity metrics have already reached a saturation point. The plot shows its variation from a bit over 0% (i.e., no toggles at all) to about 100% (i.e., all bit toggles for the best individual of the last population). The best test programs are also fault simulated (using the SAF model), and the evolution is plotted together with the connectivity (in blue). After 101 generations, we reached a fault coverage of about 90% (the targeted fault coverage) as illustrated by the vertical and dotted red line. As the generations progress, fault coverage shows a growing trend with reduced oscillations between best-identified test programs. The μGP tool evaluated a total of 7868 distinct programs, with an evaluation runtime of about 70 seconds each and 153 hours (about 6.4 days) of execution time to achieve the targeted fault coverage. Nonetheless, we let the experiment run to see if it could be further improved, and we managed to reach a fault coverage of 92.34% at generation 185, a slight increase compared to the same SBST in Table VII. Overall, μGP evaluated 14242 programs in 277 hours (i.e., about 11.5 days). With the current setup, the time taken by the on-chip evaluation is mainly used to generate the instruction trace; the trace analysis takes less than one second for each program. Thus, if a faster trace generation were implemented, the runtime would significantly reduce. In contrast, running a single-thread fault simulation campaign of the given programs would take up to 27 minutes each, over 23x longer than the proposed development flow based on the proposed methodology.

To further substantiate the claims regarding the methodology's effectiveness, we performed additional experiments on a different unit of the device, the integer divider unit. Figure 10 shows the evolution of a test program for an integer divider unit.

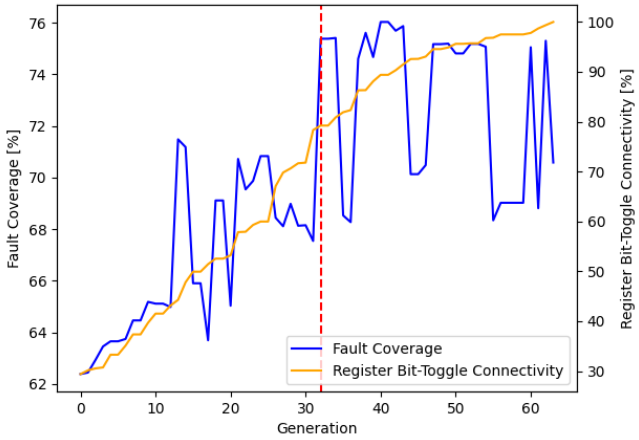


Fig. 10: Evolutionary generation of SBST for the integer divider unit.

Figure 10 shows (generation after generation) the evolution of the best programs for the population in terms of fault coverage (blue line) and the register bit-toggle connectivity. Figure 10 shows that the register bit-toggle connectivity monotonically increases with a certain regularity. On the contrary, the fault coverage oscillates in a range of about 10% even if it follows an increasing overall trend. This behavior is mainly due to the sequential pipeline nature of the module and the generated data pattern by μGP. As for the previous experiment, the plot shows that in merely 32 generations (highlighted by the vertical and red dotted line), we reach a fault coverage of about 75% (the targeted fault coverage), after evaluating 3654 programs. As for the previous experiments, we evaluate each program in about 70 seconds; thus, we require a total evaluation time of about 71 hours (i.e., 3 days). If we let the evolution run for more generations, we achieved the best program with 76.03% of fault coverage in only eight generations and about 800 evaluated programs.

Generation Strategy	Execution Time [h]	Number of individuals	Best Program Fault Coverage [%]
Baseline (FSIM)	830	1855	75.07
Proposed methodology	71	3654	75.38

TABLE IX: Comparison of different generation strategies for developing the SBST for the divider unit.

We performed the same evolution using fault coverage as a target metric to compare the achieved result with a more traditional generation of test programs. Table IX shows the results of the fault simulation-based and the connectivity-based evolutions side by side. The two methodologies reach very similar results, and the fault-simulation-based approach requires around half of the generated programs to reach the

desired fault coverage. Nonetheless, the proposed methodology requires only a fraction of the time to evaluate each individual, thus reaching the expected result much faster than the counterpart with a speedup of 8.55x.

As a final consideration, we want to highlight the possibility of fine-tuning the result reached with the connectivity metric. Using μGP we can replace the evaluator function to switch from the connectivity metric to the fault coverage, thus evolving a population composed of already promising individuals instead of starting from a population of random individuals.

V. Conclusions and Future Works

Developing functional test programs requires enormous execution times due solely to feedback from fault simulation campaigns. Moreover, it often proceeds blindly, as the fault simulator does not explicitly indicate how to modify the programs to increase the fault coverage. Consequently, the efficiency of the functional test program phase strongly depends on the experience and capability of the test engineers. In addition, for SLT applications and SDC-oriented functional test programs, executing fault simulation campaigns is unfeasible.

We propose a methodology based on analyzing the instruction trace to indirectly grade the fault detection capabilities of a functional test program, i.e., its ability to propagate all data values till signature points or the end of the program. Our connectivity metrics provide an indirect way of grading the quality of functional test programs and deliver fine-grained feedback to locate the exact code instructions affecting the correctness of the data flow. This information guides test engineers in evaluating the quality of their programs before even executing fault simulations. Due to its high flexibility, our methodology accepts different trace file formats, which can be enriched with the ISA description to capture better how data propagates between instructions.

Among future works, we would like to mention that we are extending the proposed methodology to multi and many-core systems [42], where data flow must be traced across different computing units synchronizing through specific barriers or fence instructions.

References

[1] P. Bernardi and et al., "Development flow for on-line core self-test of automotive microcontrollers," *IEEE Transactions on Computers*, vol. 65, no. 3, pp. 744–754, March 2016.

[2] S. M. Thatte and J. A. Abraham, "Test generation for microprocessors," *IEEE Transactions on Computers*, vol. C-29, no. 6, pp. 429–441, 1980.

[3] D. Brahme and J. A. Abraham, "Functional testing of microprocessors," *IEEE Transactions on Computers*, vol. C-33, no. 6, pp. 475–485, 1984.

[4] V. Gangaram, D. Bhan, and J. K. Caldwell, "Functional test selection for high volume manufacturing," in *Seventh International Workshop on Microprocessor Test and Verification (MTV'06)*, 2006, pp. 15–19.

[5] F. Angione, P. Bernardi, G. Filipponi, M. S. Reorda, D. Appello, V. Tancorre, and R. Ugioli, "An optimized burn-in stress flow targeting interconnections logic to embedded memories in automotive systems-on-chip," in *2022 IEEE European Test Symposium (ETS)*, May 2022, pp. 1–6.

[6] D. Appello, C. Bugeja, G. Pollaccia, P. Bernardi, R. Cantoro, M. Restifo, E. Sanchez, and F. Venini, "An optimized test during burn-in for automotive soc," *IEEE Design Test*, vol. 35, no. 3, pp. 46–53, 2018.

[7] D. Appello et al., "System-level test: State of the art and challenges," in *IEEE International Symposium on On-Line Testing and Robust System Design (IOLTS)*, 2021.

[8] H. H. Chen, "Beyond structural test, the rising need for system-level test," in *International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, 2018.

[9] I. Polian, J. Anders, S. Becker, P. Bernardi, K. Chakrabarty, N. El-Hamawy, M. Sauer, A. Singh, M. S. Reorda, and S. Wagner, "Exploring the mysteries of system-level test," in *2020 IEEE 29th Asian Test Symposium (ATS)*, 2020, pp. 1–6.

[10] D. K. R. Tipparthi and K. K. Kumar, "Concurrent system level test (cslt) methodology for complex system-on-chip," in *2014 IEEE 16th Electronics Packaging Technology Conference (EPTC)*, 2014, pp. 196–199.

[11] P. H. Hochschild, P. Turner, J. C. Mogul, R. Govindaraju, P. Ranganathan, D. E. Culler, and A. Vahdat, "Cores that don't count," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, ser. HotOS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 9–16. [Online]. Available: https://doi.org/10.1145/3458336.3465297

[12] H. D. Dixit, L. Boyle, G. Vunnam, S. Pendharkar, M. Beadon, and S. Sankar, "Detecting silent data corruptions in the wild," March 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2203.08989

[13] P. Bernardi, M. Grosso, E. Sanchez, and O. Ballan, "Fault grading of software-based self-test procedures for dependable automotive applications," in *2011 Design, Automation & Test in Europe*, March 2011, pp. 1–2.

[14] D. Appello, P. Bernardi, A. Calabrese, G. Pollaccia, S. Quer, V. Tancorre, and R. Ugioli, "Parallel multithread analysis of extremely large simulation traces," *IEEE Access*, vol. 10, pp. 56 440–56 457, 2022.

[15] F. Corno, G. Cumani, M. Sonza Reorda, and G. Squillero, "Efficient machine-code test-program induction," in *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No.02TH8600)*, vol. 2, May 2002, pp. 1486–1491 vol.2.

[16] F. Angione, P. Bernardi, A. Calabrese, L. Cardone, A. Niccoletti, D. Piumatti, S. Quer, D. Appello, V. Tancorre, and R. Ugioli, "An innovative strategy to quickly grade functional test programs," in *2022 IEEE International Test Conference (ITC)*, 2022, pp. 355–364.

[17] G. Papadimitriou and D. Gizopoulos, "Silent data corruptions: Microarchitectural perspectives," *IEEE Transactions on Computers*, vol. 72, no. 11, pp. 3072–3085, 2023.

[18] F. Angione, P. Bernardi, N. di Gruttola Giardino, D. Appello, C. Bertani, and V. Tancorre, "A guided debugger-based fault injection methodology for assessing functional test programs," 2023.

[19] A. Hanneman, J. Gava, V. Bandeira, R. Garibotti, R. Reis, and L. Ost, "Debate-fi: A debugger-based fault injector infrastructure for iot soft error reliability assessment," in *2023 IEEE 9th World Forum on Internet of Things (WF-IoT)*, 2023, pp. 1–6.

[20] F. Angione, D. Appello, P. Bernardi, A. Calabrese, S. Quer, V. Tancorre, and R. Ugioli, "A toolchain to quantify burn-in stress effectiveness on large automotive system-on-chips," *IEEE Access*, 2023.

[21] F. F. d. Santos, J. E. R. Condia, L. Carro, M. S. Reorda, and P. Rech, "Revealing gpus vulnerabilities by combining register-transfer and software-level fault injection," in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2021, pp. 292–304.

[22] P. Omland, A. Netti, Y. Peng, A. Baldovin, M. Paulitsch, G. Espinosa, J. Parra, G. Hinz, and A. Knoll, "Hpc hardware design reliability benchmarking with hdfit," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 3, pp. 995–1006, 2023.

[23] P. Bernardi, M. Grosso, E. Sanchez, and O. Ballan, "Fault grading of software-based self-test procedures for dependable automotive applications," in *2011 Design, Automation & Test in Europe*, March 2011, pp. 1–2.

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

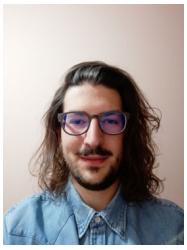
[25] M. Grosso *et al.*, "A software-based self-test methodology for system peripherals," in *IEEE ETS*, 2010, pp. 195–200.

[26] A. Paschalis and D. Gizopoulos, "Effective software-based self-test strategies for on-line periodic testing of embedded processors," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 1, pp. 88–99, 2005. [Online]. Available: https://doi.org/10.1109/TCAD.2004.839486

[27] P. K. Parvathala and et al., "Functional random instruction testing (frits) method for complex devices such as microprocessors," *United States Patent 6948096*, 2005.

[28] D. Piumatti and et al., "An efficient strategy for the development of software test libraries for an automotive microcontroller family," *Microelectronics Reliability*, vol. 115, dec 2020.

[29] T. Mak, "Infant mortality–the lesser known reliability issue," in *13th IEEE International On-Line Testing Symposium (IOLTS 2007)*, 2007, pp. 122–122.
[30] P. Bernardi *et al.*, "Applicative system level test introduction to increase confidence on screening quality," in *IEEE DDECS*, 2020, pp. 1–6.
[31] M. Hassan *et al.*, "Early soc security validation by vp-based static information flow analysis," in *IEEE/ACM ICCAD*, 2017.
[32] R. Drechlser *et al.*, "Ensuring correctness of next generation devices: From reconfigurable to self-learning systems," in *IEEE ATS*, 2019.
[33] W. Hu *et al.*, "An overview of hardware security and trust: Threats, countermeasures, and design tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2020.
[34] K. M. Alatoun *et al.*, "Efficient methods for soc trust validation using information flow verification," in *IEEE ICCD*, 2021.
[35] G. Cabodi, P. Camurati, and S. Quer, "Symbolic Exploration of Large Circuits with Enhanced Forward/Backward Traversals," in *Proc. IEEE EURO–DAC'94*. Grenoble, France: IEEE Computer Society, sep 1994, pp. 22–27.
[36] G. Cabodi, S. Nocco, and S. Quer, "Mixing Forward and Backward Traversals in Guided-Prioritized BDD-Based Verification," in *Proc. Computer Aided Verification*, ser. Lecture Notes in Computer Science, E. M. Clarke and R. P. Kurshan, Eds., vol. 2102. Copenhagen, Denmark: Springer-Verlag, jul 2002, pp. 471–484.
[37] A. Ressel and R. Schmidt-Vollus, "Reverse engineering in process automation," in *2021 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2021, pp. 1–4.
[38] A. K. Dwivedi, S. K. Rath, S. M. Satapathy, L. S. Chakravarthy, and P. K. S. Rao, "Applying reverse engineering techniques to analyze design patterns in source code," in *2018 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, 2018, pp. 1398–1404.
[39] E. C. Michael J. Eager, "Introduction to the dwarf debugging format," https://dwarfstd.org/doc/Debugging%20using%20DWARF-2012.pdf, accessed: 18 July 2024.
[40] N. di Gruttola Giardino, F. Angione, P. Bernardi, T. Foscale, C. Bertani, and V. Tancorre, "A flexible fpga-based test equipment for enabling out-of-production manufacturing test flow of digital systems," in *2024 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, 2024, pp. 1–6.
[41] J. J. Labrosse, *UC/OS-III, The Real-Time Kernel, or a High Performance, Scalable, ROMable, Preemptive, Multitasking Kernel for Microprocessors, Microcontrollers & DSPs*. Weston, FL, USA: Micrium Press, 2009.
[42] A. Borione, L. Cardone, A. Calabrese, and S. Quer, "An experimental evaluation of graph coloring heuristics on multi- and many-core architectures," *IEEE Access*, vol. 11, pp. 125 226–125 243, 2023.



Francesco Angione is a Computer Engineer with an M.Sc. in Embedded Systems track, obtained from Politecnico di Torino in 2020. He is working towards his Ph.D. in "System-Level-Test Techniques for Automotive SoCs" at Politecnico di Torino in the CAD & Reliability group. His main interests are real-time operating systems, computer architectures, and their dependability.

Paolo Bernardi (MS'02 and PhD'06 in Computer Science) is an Associate Professor of Politecnico di Torino University, working in the Electronic CAD and Reliability research group. His interests include system-on-chip tests and reliability, especially in high-quality automotive devices. Prof. Bernardi is the Program Chair of the Automotive Reliability, Test, and Safety (ARTS) Workshop held in conjunction with the International Test Conference and General Chair of the IEEE European Test Symposium 2023. Paolo Bernardi is an IEEE senior member.

Andrea Calabrese earned a Ph.D. in Computer Science from Politecnico di Torino in 2024. Currently, he is employed at Amarula Solutions in the Netherlands, focusing on software-hardware interaction testing between different devices.

Lorenzo Cardone graduated in Computer Science from Politecnico di Torino in 2021 and he is currently working with a research grant with the same university. His main field of research is software optimization and parallel algorithms.

Stefano Quer received an M.S. in Electronic Engineering in 1991, and a Ph.D. in Computer Engineering in 1996. He has been a Visiting Faculty in the Department of Electronic Engineering and Computer Science at the University of California in Berkeley, an intern with the "Advanced Technology Group" at Synopsys Inc., and with the "Alpha Development Group" at Compaq Computer Corporation. He has been a Compaq Computer Corporation consultant. He is currently a professor with the Department of Control and Computer Engineering at Politecnico di Torino, Italy. His main research interests include systems and tools for CAD for VLSI, formal methods for hardware and software systems, and the development of sequential and concurrent algorithms that can achieve acceptable solutions with limited resources.

Claudia Bertani is a Product & Test Engineer Manager at STMicroelectronics. She graduated in Electronic Engineering at Politecnico di Milano. She has 20+ years of expertise in the semiconductor industry, high-volume product ramp-up and management, semiconductor manufacturing flow, DfT, and testability analysis, ATE, new product introduction technical support, and team management. She manages the team in charge of System-Level Test introduction on Automotive ADAS SOC products.

Vincenzo Tancorre is a Yield Enhancement engineer in the Automotive Group at STMicroelectronics. His research interests include test-related process monitoring using diagnostic solutions for memories and unstructured logic based on DfT methodologies. Tancorre has a BS in electronics engineering from Politecnico di Bari.