Non-Intrusive Self-Test Library for Automotive Critical Applications: Constraints and Solutions

P. Bernardi[1], R. Cantoro[1], A. Floridia[1], D. Piumatti[1], C. Pogonea[1], A. Ruospo[1], E. Sanchez[1]
S. De Luca[2], A. Sansonetti[2]

[1] *Politecnico di Torino – Dipartimento di Automatica e Informatica - Italy*
[2] *STMicroelectronics, Italy*

***Abstract* — Today, safety-critical applications require self-tests and self-diagnosis approaches to be applied during the lifetime of the device. In general, the fault coverage values required by the standards (like ISO 26262) in the whole System-on-Chip (SoC) are very high. Therefore, different strategies are adopted. In the case of the processor core, the required fault coverage can be achieved by scheduling the periodical execution of a set of test programs or Software-Test Library (STL). However, the STL for in-field testing should be able to comply with the operating system specifications without affecting the mission operation of the device application. In this paper, the most relevant problems for the development of the STL are first discussed. Then, it presents a set of strategies and solutions oriented to produce an efficient and non-intrusive STL to be used exclusively during the in-field testing of automotive processor cores. The proposed approach was experimented on an automotive SoC developed by STMicroelectronics.**

Keywords — Software Test Library, Non-Intrusive Test, Self Test

I. INTRODUCTION

Automotive electronic systems are commonly labeled as safety-critical systems. As a consequence, car manufactures need to guarantee a high reliability level on the different electronics components of the vehicle. One of the techniques commonly adopted to increase the device functional safety capabilities is the in-field testing. In-field testing can be supported by the insertion of specific hardware modules in charge for monitoring the different elements in the device during the power-on or power-off [1], such as the processor core [2] or the memory [3] healthy; or for example, concurrently testing during the normal mode operation that no faults affected the memory by including ECC modules [4] [5]. Hardware-based mechanisms correctly adopted may guarantee high coverage values, though these are invasive and costly solutions. On the other side, in-field testing can be supported by software-based solutions or Software-Based Self-Test (SBST); in this case, the processor core is asked to execute very carefully devised test programs, in automotive usually called Software Test Library (STL), during the power-on, power-off, or periodically. The target is to test the processor or some peripherals around it. This strategy, initially proposed by [6], has been studied by different research groups [15][16] as described in [8], and later extended targeting the automotive field in [7]. Main advantages of SBST techniques are: *at-speed testing* capabilities and it does not require additional hardware within the final device, even though it is necessary to allocate some Flash memory space to store the software library. Unfortunately, the test program creation is still a very time-consuming process**.** Today many semiconductor and IP companies are widely adopting these techniques, for example: ARM [9], Infineon [10], STMicroelectronics [11], Cypress [12], Renesas [13], and Microchip [14]. STLs are usually composed of *intrusive* and *non-intrusive* test programs. The former set (also referred to as *non-exceptive* and *critical* tests [7]) is usually executed during the power-on and power-off to avoid conflict with the Operating System (OS). Indeed, supervisor capabilities are needed because these programs manipulate special registers, trigger exceptions, and so on. The latter set of programs instead (also known as *run-time* tests [7]), can be executed by the OS as a simple application requiring no special conditions. Although this coexistence simplifies the code memory allocation, it introduces several limitations. For instance, non-intrusive tests are not allowed to freely write all the available RAM to avoid overwriting its content. Generally speaking, as described in [7], a test program must be carefully structured, to configure memory areas and peripheral resources for test purposes. Then, to interact with the OS code, the test program must comply with an Embedded-Application Binary Interface (EABI) [17]. Finally, each test program produces a result called *signature*. This is compared with a stored one and, in case these differ, the processor is considered as faulty. In this paper, the most relevant characteristics necessary to develop a set of non-intrusive test programs are presented as well as a series of strategies to divert already existing intrusive test programs algorithms in non-intrusive ones. The experimental results were gathered on a dual issue processor core manufactured by STMicroelectronics targeting automotive applications.

II. NON-INSTRUSIVE CONSTRAINTS

In this section, the constraints that must be satisfied to write a non-intrusive test are indicated and discussed, along with the processor units that can be tested**.** Non-intrusive tests must not modify, nor alter the state of the system in order not to affect the OS or the application code. The constraints each test must comply with are:

No alteration of Special-Purpose Registers (SPRs). Since SPRs are used to configure the core, the management of these registers is reserved to the OS.

No alteration of RAM memory. The RAM memory is managed by the Operating System as system memory.

Modifying this memory may involve an alteration of the operation of the application code.

Maximum execution time. Automotive applications are often executed in real time. This means that the time slot (which is application dependent) reserved for each test is quite limited. This time slot represents the maximum test execution time.

No interruptions, nor exceptions. The Exception unit is configured to manage different sources of interrupt. One or more System-Interrupt Service Routine (S-ISR) is written for satisfying each specific request. A test program for the exception unit makes use of intentionally raised interrupts and exceptions managed by a Test-Interrupt Service Routine (T-ISR) as an alternative to the S-ISR. Since ISRs are part of the OS, this modification is not allowed.

The test must be interruptible. A test must be interruptible by an interrupt request. This implies: 1) registers devoted to the context switch should not be corrupted; 2) maintain unaltered the execution of the user code. This is normally achieved by writing test program compliant to the EABI [17] standard.

No peripheral usage. Non-intrusive tests cannot use or configure peripherals, as their use would change the general behavior of the SoC.

User Mode Configuration. Application code is normally executed in User Mode (i.e., with limited access to the system resources). The non-intrusive STL is executed concurrently with the application software, thus the execution must be in User Mode too.

III. PROPOSED NON-INTRUSIVE TEST ALGORITHMS

This section proposes the strategy to transform some of the intrusive tests algorithms into new non-intrusive tests. According to [7], the units inside a microcontroller could be split into five categories such as: *Arithmetic units* dedicated to execution of specific arithmetic or logical operations. *Special units* dedicated to the management of memories, exceptions and load-store operations. *Register File unit* containing all General-Purpose Register and Special-Purpose Register. *Addressing units* that include the calculation of the Effective Address, the management of Program Counter and the Branch Prediction unit (BPU). Finally, *Control modules* devoted to managing the pipeline, forwarding paths and decoding the opcode of an instruction. The following describes in detail the techniques and the algorithms that have been used to create non-intrusive tests for some units or subunits of those mentioned above.

Arithmetic Unit: When dealing with these modules, self-test procedures are often developed resorting to an Automatic Test Pattern Generation (ATPG) tool [18]. Basically, the test patterns generated by the tool are treated as operands for arithmetic unit, and such operands are then fed to the unit by means of suitable assembly instructions. Creating a non-intrusive test program from an intrusive one requires only fragmenting the original test program into many sub-test programs, whose duration complies with the non-intrusive constraints.

Load-Store Unit: Load-Store units are usually tested resorting to test programs that use specific RAM memory addresses. An effective technique consists in choosing a target address in the memory and performing load and store operations with checkerboard patterns and compacting the result of the test into a signature. In the first part of the algorithm, a store-word operation of the pattern 0101…0 is executed, followed by 4 load-byte operations and, finally, by 2 load-half-word operations. The sequence of operations is then repeated using a complementary pattern. In the second part of the algorithm, 4 store-byte operations to adjacent addresses are executed, followed by a final load-word operation. Again, the sequence is then repeated using a complementary pattern. The non-intrusive version of the algorithm described above can be created by: 1) subdividing the operations into multiple sub-tests; 2) executing the load and store operations using exclusively the stack frame.

Register File: An effective algorithm to test the Register File unit is described in [19]. The approach organizes registers into groups, according to the hamming distance between registers' encodings. This algorithm can be applied both to GPRs and SPRs. In the non-intrusive version of the test algorithm, only GPRs can be tested. Moreover, the whole test algorithm may be subdivided into multiple sub-tests to meet the maximum execution time constraint.

BPU: In this case, a BTB is considered. This unit can be effectively tested using a March Test algorithm proposed in [20]. The algorithm should locate jump instructions into very specific memory locations spread all over the RAM memory to test all the stuck-at faults associated to each bit of the table. An effective solution is to write such jump instructions dynamically into the desired addresses, and then to perform a jump to that addresses to let the processor fetch instructions and store addresses into the BTB. The number of memory locations used is equal to the number of BTB entries. Since this algorithm easily exceeds the maximum execution time constraint, a non-invasive version of the algorithm can be created by carefully subdividing it into multiple sub-tests. The number of jumps that can be executed in a sub-test depends on the maximum execution time constraint and can be very critical. In the worst case, the execution time may be sufficient for a couple of jump operations, that is, to load one entry of the BTB. The side-effect of splitting the algorithm into sub-tests is a non-deterministic behavior in the BTB entries due to the actual values in all the entries in the prediction unit, and consequently a non-deterministic fault coverage. Since the Branch Prediction Unit (BPU) is not initialized before running the test and the replacement policy adopted by BTBs is the Least-Frequently Used (LFU), each sub-test should be repeated multiple times to cover all entries. Moreover, since the candidate for the replacement is not known in advance at runtime, the problem must be tackled statistically, using the probabilistic theorem of geometric distributions. For the geometric distributions, given a geometrically distributed random variable X, the *expected value* $E(X)$ is computed using the formula $E(X) = 1/p$. In this formula, X is the number of trials up to (and including) the first success, which in our case is the number of times needed to run a program to hit an entry different than the previous one, while p is the probability of success, i.e., the probability to hit a different entry. The average number of times that the test

program should be executed to use and cover all the BTB entries can be computed using the formula reported in Equation 1, where n is the number of BTB entries. As an example, let us consider an 8-entry BTB: the probability of success for the first entry is 1, for the second is $7/8$, for the third is $6/8$, and so on; according to the formula, each sub-test should be executed 22 time to statistically cover all the entries.

$$\sum_{i=1}^{n} E(X_i) = 1 + \frac{n}{(n-1)} + \cdots + \frac{n}{(n-(n-1))} = n \sum_{i=1}^{n} \frac{1}{i}$$

Equation 1: Expected value of a random variable

The second problem of the algorithm concerns to the RAM memory addresses that can be used. The test program cannot be freely allocated into the program memory since it must coexist with the user application code (which is not known a priori). The only viable solutions to increase the variability of the addresses used (and thus the fault coverage) is to replicate the sub-test code multiple times in the program memory.

Decode unit: The Decode unit can be tested using the approach proposed in [21], which makes use of the user manual to derive the list of legal assembly instructions and derives specific illegal instructions to complement the test. The principle of the approach is to identify instructions whose opcodes are 1-bit hamming distance far from each other, such that a deviation in a test instruction produces a corruption of the test signature. For instance, let us consider the two instructions in Table I, whose opcodes differ in the bit 29. The first instruction executes an arithmetic sum, while the second one loads an immediate value into a register. When executing these two instructions one after the other, a stuck-at fault on the bit 29 transforms one of the two instructions into the other, performing a different operation from expected one. The same reasoning is applied to each bit of the opcode.

TABLE I. LEGAL INSTRUCTION TESTING FOR DECODER UNIT

Opcode bit test	First legal instruction		Second legal instruction	
	opcode	Mnemonic	opcode	mnemonic
29	7C000214	add r0 r0 r0	3C000214	lis r0, 0x124

This approach uses the same procedure to find pairs of near instructions in which one of the two opcodes is illegal. However, when an illegal instruction is executed, an internal exception is raised. Since it does not comply with the constraint on interruptions, the non-invasive version of the program can only make use of pairs of legal instructions. Therefore, the whole test program must be cut into several sub-tests to respect the required time constraints.

IV. EXPERIMENTAL RESULTS

As case study, it was selected a Zen Z446 processor embedded in an industrial SoC manufactured by STMicroelectronics, tailored for safety-critical automotive applications. It consists of a 32-bit in-order dual-issue five-stage pipeline, an 8-entry BTB, instruction fetch, multi-port register file, and load and store unit. The post-layout netlist of the CPU accounts for 800k stuck-at faults. The fault coverage of the developed STL was computed by means of fault injection campaigns, leveraging a commercial fault simulator. All the non-intrusive constraints discussed in Section 3 were applied to the case study. Specifically, each test was constrained to 255 clock cycles. This value stems from the specifications provided by ST Microelectronics. Table III highlights the characteristics of the final STL, when considering an intrusive (i.e., unconstrained) approach. For each module within the CPU, the number of self-test procedures for its test are reported, along with the relevant characteristics (namely, duration and memory footprint).

TABLE III. CHARACTERISTCS INTRUSIVE STL

CPU Module	***# test programs***	***Duration* [cc.]**	***Size* [KB]**
Arithmetic adder	3	988	1.1
Divider	4	4,168	1.4
Logic operation	11	5,625	2.7
Multiplier	2	960	0.9
Shifter	3	820	1.5
Exception	1	8,400	1.8
BTB	3	2,916	2.4
Register file	1	1,800	2.1
Fetch unit	1	30,900	1.5
Forward unit	1	3,400	4
Decoder unit	1	1,500	2.7
Load / Store unit	2	750	1.3
TOTAL	**33**	**62,227**	**23.4**

The same units are reported in Table IV for the non-intrusive approach. It is tangible that total number of test programs is higher compared to the intrusive one. This is reasonable, since the chosen time constraint of 255 clock cycles is quite stringent and it forces to separate and eventually adapt each test into smaller chunks to fit that time slot. It is also important to note that some test programs (namely, Exception, Fetch Unit, and Forward) cannot be implemented in the non-intrusive approach due to the constraints. Therefore, the fault coverage related to the modules they cover is exclusively a cascade effect [7].

TABLE IV. CHARACTERISTCS NON-INTRUSIVE STL

CPU Module	***# test programs***	***Duration* [cc.]**	***Size* [KB]**
Arithmetic adder	6	1,309	1.3
Divider	21	4,763	2.1
Logic operation	28	6,163	3.6
Multiplier	5	1,055	1.2
Shifter	4	714	1.7
Exception	0	0	0
BTB	28	6,975	8.5
Register file	4	927	1.7
Fetch unit	0	0	0
Forward unit	0	0	0
Decoder unit	5	970	1.4
Load / Store unit	4	890	1.5
TOTAL	**105**	**23,766**	**23**

Moreover, although the test programs were split into different chunks, the overall execution time of some of them (e.g., Arithmetic Adder and Multiplier) is higher in the non-intrusive version than in the intrusive one, due to the overhead of saving and restoring all the used registers. Regarding the BTB test program, after several experiments, it was found that the optimal number of test programs was 28, that is 4 sub-tests replicated 7 times in Flash. The programs were allocated in Flash at random addresses, to mimic the case in which also the

user application code is present in Flash (which is not known a priori). Although the number of test programs in the non-intrusive version is higher (105 against 33), the number of clock cycle required for executing the entire STL is lower compared to the intrusive one (23,766 against 62,227), because the most intrusive test program (e.g., those that trigger interrupts) cannot be executed at all (such programs are also the most time-consuming ones). The same applies to the overall memory footprint, which is comparable for both versions. Table V compares the fault coverage achieved for each relevant unit of the CPU in both cases: intrusive test and non-intrusive one.

TABLE V. FAULT COVERAGE INTRUSIVE VERSUS NON-INTRUSIVE STL

CPU Module	***# faults***	***FC Intrusive* [%]**	***FC Non-intrusive* [KB]**
Arithmetic adder	10,268	88.39%	87.50%
Divider	20,625	82.37%	80.46%
Logic operation	19,266	83.83%	82.23%
Multiplier	71,541	96.58%	96.46%
Shifter	12,548	96.64%	95.68%
Exception	13,695	37.36%	7.97%
BTB	29,677	68.41%	53.64%
Register file	264,022	91.59%	68.22%
Address generator	28,419	61.37%	43.56%
Fetch unit	58,244	62.73%	59.26%
Forward unit	103,506	67.26%	66.69%
Decoder unit	91,523	56.84%	45.40%
Load / Store unit	12,420	89.71%	89.53%
Control unit	50,442	56.93%	55.81%
TOTAL	**79,7562**	**75.31%**	**67,12%**

The reported data were filtered considering that 1.63% of the total faults were labeled as untestable, following the guidelines presented in [22]. Clearly, a non-invasive solution can obtain a lower fault coverage with respect to the intrusive one due to the constraints previously described. By processing the final full fault lists, given the constraints, it emerged that the percentage of faults that cannot be excited by the self-test procedures accounts for the 8.19% of faults of the full fault list (which is the difference in terms of fault coverage between the two STLs). It is worth noting that those faults can be addressed with an intrusive test, only. Indeed, they result being detected with the intrusive test programs. Therefore, they cannot be labeled as untestable and removed from the final fault coverage. Finally, one of the units that negatively affect the fault coverage in the non-intrusive test programs is the register file since many registers are accessible exclusively in supervisor mode.

V. CONCLUSIONS

The end-goal of this paper is to present the most relevant constraints when developing an STL for device deployed in safety-critical scenario. Guidelines were provided for developing a non-intrusive STL, which complies with a given set of constraints. Such guidelines are also intended to make the STL as transparent as possible to the operating system, which is supposed to handle the different tasks (i.e., both user and test programs). Cleary, the aforementioned constraints can be relaxed if the operating system is aware and capable of handling such self-test procedures, allowing for a more intrusive test.

VI. REFERENCES

[1] G. Tshagharyan, G. Harutyunyan and Y. Zorian, "An effective functional safety solution for automotive systems-on-chip", 2017 IEEE International Test Conference (ITC), Fort Worth, TX, 2017, pp. 1-10.

[2] T. McLaurin, "Periodic Online LBIST Considerations for a Multicore Processor," *2018 IEEE International Test Conference in Asia (ITC-Asia)*, Harbin, China, 2018, pp. 37-42.

[3] D. Sargsyan, Y. Zorian, "ISO 26262 compliant memory BIST architecture", 2017 Computer Science and Information Technologies (CSIT), 2017, pp. 78-82.

[4] E. Fujiwara, "Code Design for Dependable Systems: Theory and Practical Application," Ed. Wiley-Interscience, 2006.

[5] C. L. Chen and M. Y. Hsiao, "Error-Correcting Codes for Semiconductor Memory Applications: A State-of-the-Art Review," in IBM Journal of Research and Development, vol. 28, no. 2, pp. 124-134, March 1984.

[6] M. Thatte and J.A. Abraham, "Test Generation for Microprocessors", IEEE Trans. Computers, vol. 29, no. 6, 1980, pp. 429-441

[7] P. Bernardi, R. Cantoro, S. De Luca, E. Sánchez and A. Sansonetti, "Development Flow for On-Line Core Self-Test of Automotive Microcontrollers," in *IEEE Transactions on Computers*, vol. 65, no. 3, pp. 744-754, 1 March 2016.

[8] M. Psarakis, D. Gizopoulos, E. Sanchez and M. Sonza Reorda, "Microprocessor Software-Based Self-Testing," in *IEEE Design & Test of Computers*, vol. 27, no. 3, pp. 4-19, May-June 2010. doi: 10.1109/MDT.2010.5

[9] https://developer.arm.com/technologies/functional-safety

[10] https://www.hitex.com/software-components/selftest-libraries-safety-libs/pro-sil-safetcore-safetlib/

[11] http://www.st.com/content/ccc/resource/technical/document/application_note/02/1a/91/78/e4/15/4d/35/CD00290100.pdf

[12] http://www.cypress.com/file/249196/download

[13] https://www.renesas.com/en-eu/products/synergy/software/add-ons.html#read

[14] http://ww1.microchip.com/downloads/en/DeviceDoc/52076a.pdf

[15] A. Paschalis, D. Gizopoulos, N. Kranitis, M. Psarakis and Y. Zorian, "Deterministic software-based self-testing of embedded processor cores," *Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001*, Munich, Germany, 2001, pp. 92-96.

[16] A. Jasnetski, R. Ubar and A. Tsertov, "On automatic software-based self-test program generation based on high-level decision diagrams," *2016 17th Latin-American Test Symposium (LATS)*, Foz do Iguacu, 2016, pp. 177-177.

[17] S. Sobek and K. Burke. (2004). PowerPC Embedded Application Binary Interface (EABI): 32-Bit Implementation [Online]. Available: http://www.nxp.com/docs/application-note/PPCEEABI.pdf

[18] P. Bernardi, S. De Luca, D. Piumatti, S. Regis, E. Sanchez and A. Sansonetti, "On the in-field testing of spare modules in automotive microprocessors," 2017 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC), Abu Dhabi, 2017, pp. 1-6.

[19] P. Bernardi, R. Cantoro, S. De Luca, E. Sanchez, A. Sansonetti and G. Squillero, "Software-Based Self-Test Techniques for Dual-Issue Embedded Processors," in IEEE Transactions on Emerging Topics in Computing.

[20] D. Changdao, M. Graziano, E. Sanchez, M. Sonza Reorda, M. Zamboni and N. Zhifan, "On the functional test of the BTB logic in pipelined and superscalar processors," 2013 14th Latin American Test Workshop - LATW, Cordoba, 2013, pp. 1-6.

[21] P. Bernardi et al., "On the in-field functional testing of decode units in pipelined RISC processors," 2014 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), Amsterdam, 2014, pp. 299-304.

[22] P. Bernardi, M. Bonazza, E. Sanchez, M. Sonza Reorda and O. Ballan, "On-line functionally untestable fault identification in embedded processor cores," *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Grenoble, France, 2013, pp. 1462-146.