POLITECNICO DI TORINO
Repository ISTITUZIONALE

Deterministic Cache-based Execution of On-line Self-Test Routines in Multi-core Automotive System-on-Chips

Original
Deterministic Cache-based Execution of On-line Self-Test Routines in Multi-core Automotive System-on-Chips / Floridia, A., Carmona, T.M., Piumatti, D., Ruospo, A., Sanchez, E., Luca, S.D., Martorana, R., Pernice, M.A.. - ELETTRONICO. - (2020), pp. 1235-1240. (2020 Design, Automation & Test in Europe Conference & Exhibition (DATE) Grenoble, France 9-13 March, 2020) [10.23919/DATE48585.2020.9116239].

Availability:
This version is available at: 11583/2837295 since: 2020-06-25T12:19:30Z

Publisher:
IEEE

Published
DOI:10.23919/DATE48585.2020.9116239

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright
IEEE postprint/Author's Accepted Manuscript

©2020 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collecting works, for resale or lists, or reuse of any copyrighted component of this work in other works.

(Article begins on next page)

25 November 2026

Deterministic Cache-based Execution of On-line Self-Test Routines in Multi-core Automotive System-on-Chips

Andrea Floridia*, Tzamn Melendez Carmona*, Davide Piumatti*, Annachiara Ruospo*, Ernesto Sanchez*
Sergio De Luca†, Rosario Martorana†, Mose Alessandro Pernice †
*Dipartimento di Automatica e Informatica, Politecnico di Torino, Italy
†STMicroelectronics, Italy

***Abstract*—Traditionally, the usage of caches and deterministic execution of on-line self-test procedures have been considered two mutually exclusive concepts. At the same time, software executed in a multi-core context suffers of a limited timing predictability due to the higher system bus contention. When dealing with self-test procedures, this higher contention might lead to a fluctuating fault coverage or even the failure of some test programs. This paper presents a cache-based strategy for achieving both deterministic behaviour and stable fault coverage from the execution of self-test procedures in multi-core systems. The proposed strategy is applied to two representative modules negatively affected by a multi-core execution: synchronous imprecise interrupts logic and pipeline hazard detection unit. The experiments illustrate that it is possible to achieve a stable execution while also improving the state-of-the-art approaches for the on-line testing of embedded microprocessors. The effectiveness of the methodology was assessed on all the three cores of a multi-core industrial System-on-Chip intended for automotive ASIL D applications.**

I. Introduction

Test solutions based on the usage of Software Test Libraries (STLs) are increasingly becoming adopted for the on-line testing of automotive processor-based System-on-Chips (SoCs) [1]–[7]. They are based on a set of software self-test procedures, intended for detecting the occurrence of possible permanent faults affecting the processor core. The main idea of this approach (initially proposed in [8]) is to convert test patterns into software instructions and accumulate their results to create a so-called test signature. Then, such a signature is compared with the expected test signature (obtained in a fault-free scenario) to determine whether the test passed or failed. When the test is executed in field, the test signature represents *the only way to safely detect the occurrence of faults* [9]–[12]. Self-test procedures can be broadly distinguished into two main categories [2], [7]: boot-time and run-time tests. The former are executed during the namesake phase of the device, when it is entering the on-line phase. The latter are executed concurrently with the application software. Some boot-time test programs, in order to be effective, require a proper sequence of instructions to be executed without any interruption. Moreover, the STL must comply with the typical requirements of the embedded software, since it coexists with an Operating System or an application program. Therefore, the resources usage (namely code and data memory) should be minimal. The vast majority of the existing Software-Based Self-Test (SBST) techniques [10] were conceived considering exclusively a single-core execution. However, high-performance multi-core System-on-Chips are nowadays massively deployed in automotive applications. When dealing with these systems, to increase the system availability, parallel test is highly desirable. The run-time tests can be executed in parallel, usually during the processor idle times. On the other hand, the boot-time tests require special considerations [13] due to the shared portion of system RAM devoted to the test and the fact that cannot be interrupted. In particular, this last assumption cannot be guaranteed anymore. Indeed, the embedded software running in a multi-core context suffers of a limited timing predictability [14], due to the higher system bus contention. These conflicts on the system bus generate stalls when fetching instructions from the main memory and thus *the exact stream of instructions entering the pipeline cannot be determined in advance anymore*. In a multi-core SoC, this has two important consequences on the self-test procedures requiring a specific sequence of instructions. The first one concerns the fault grading: the fault coverage is *uncertain* and it might vary depending on which portion of the processor is excited due to the system bus activity. Because of this, a given fault location might not be excited correctly and therefore remains undetected. The second one is related to the signature generated by the test program, which is now *unstable*. It means that the self-test procedure cannot safely identify whether the mismatch in the signature is due to the occurrence of a fault or due to an unexpected instructions stream.

The novelty of this paper consists in the establishment of a deterministic methodology for executing in-field self-test routines in a multi-core scenario. The proposed method, based on cache memories, guarantees stable signature and deterministic fault coverage of those test routines (targeting specific CPU modules) negatively affected by a multi-core execution. The methodology does not require significant modifications of the already-existing algorithms and it does not introduce penalties from the memory footprint perspective. Along with these advantages, it does not require any additional on-chip resources.

The usage of caches has been explored to store the self-

test procedures intended for end-of-manufacturing testing of processors within a shared-memory multi-core system [15]. The purpose of that work was to reduce the test application time, avoiding off-chip memory accesses. The method is applicable exclusively for end-of-manufacturing, since it assumes that the self-test procedures are loaded into the caches through an external tester (which is not available when in field). Similarly, in [16] it was shown that a cache-aware test scheduler can take advantage of the memory hierarchy for speeding-up the run-time tests. Differently from these related works, the proposed approach deals with the in-field execution of boot-time procedures, and it uses caches for addressing the uncertainties introduced by a multi-core architecture.

Finally, the parallel execution of boot-time tests was analyzed in [13]. The paper presents some scheduling alternatives, exclusively considering the possible existing conflicts due to shared resources without properly addressing the determinism of the self-test procedures themselves in a multi-core scenario.

The paper is structured as follows: Section II describes the main issues arising from a multi-core execution. Section III presents the proposed strategy. Section IV describes the experimental results gathered on an industrial design. Section V concludes the paper, outlining future directions.

II. Multi-core Issues

The aim of this section is to introduce the main issues related to the execution of a Software Test Library (STL) in a multi-core scenario. The STL is historically considered an on-line test solution targeting exclusively the processor core. However, when deploying the STL in a multi-core system, the situation radically changes. Specifically, the behavior of some boot-time self-test procedures cannot be guaranteed anymore, producing unforeseen outcomes. In this context, as the experiments of Section IV confirm, a given self-test procedure might produce a *wrong signature* or an *uncertain fault coverage*. The former inhibits the self-diagnostic capabilities of the test program when in field. The latter concerns the fault grading, since it is not possible to guarantee a given fault coverage. This represents a serious concern, since modern functional safety standards impose stringent quality requirements (e.g., the ISO 26262 for the automotive domain).

For clarity, let us consider a simple but yet effective example: the forwarding mechanism of the classical 5-stage pipeline DLX processor and a self-test routine for testing such mechanism.

The reported example considers forwarding among two consecutive instructions. However, the reader should note that the same reasoning is perfectly applicable even to more complex multiple-issue processors. The only difference is that the forwarding can also take place among two consecutive issue packets. Let us focus on the following forwarding path: the *EX* to *EX* path that fed the processor adder. Figure 1a shows a portion of the assembly code testing the aforementioned path, along with its evolution across the pipeline stages in a single-core scenario. In this case, the forwarding mechanism is excited correctly. The second *add* instruction enters the

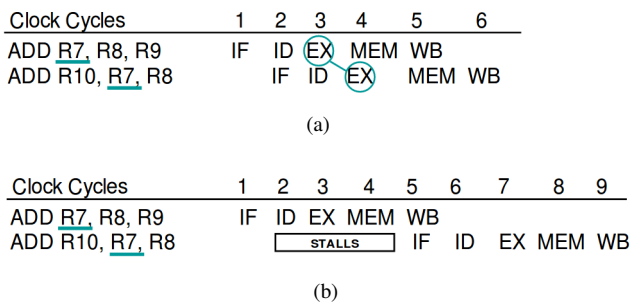


Fig. 1. Forwarding path (a), Broken forwarding path (b).

pipeline exactly one clock cycle after the first one, since the memory subsystem has not produced any stall. In order to detect the occurrence of *performance faults* [17] the processor Performance Counters can be exploited (when available). When testing these mechanisms, Performance Counters that count the number of pipeline stalls are often used since they could ease the detection of malfunctions in the hazard detection unit (e.g., stalls inserted between instructions when not needed). Figure 1b represents still the same code fragment, but in a quite different scenario.

It is assumed that the processor is part of a larger multi-core system, and the self-test procedure is executed in parallel by the other cores. As a result of the other processors' activities, the accesses to the memory subsystem are delayed. As depicted in that figure, the forwarding mechanism is not triggered at all. The second *add* enters the pipeline at the fifth clock cycle and can retrieve the content of *R7* directly from the register file, without exercising the forwarding path. This is a possible scenario that a self-test routine might encounter when executed in parallel in a multi-core SoC. In this context, the self-test procedure yields two possible outcomes:

- Wrong signature;
- Exact signature but lower fault coverage.

When the Performance Counters contribute to the signature, it is likely that the test program will produce an unstable signature. Considering the examples of Figure 1a and 1b, the execution time is slightly increased in 1b due to the additional stalls, but yet enough for altering the values of the Performance Counters, that will report 3 additional stalls. Once again, these stalls are completely unpredictable and consequently also the signature. However, it might happen that the self-test procedure does not use the Performance Counters, making the signature stable. One could erroneously believe that the test program still performs correctly as expected: however, although it returns a correct signature, the fault coverage is likely to be quite different (specifically, lower than the expected one) considering that some processor portions are not exercised. Indeed, in the scenario depicted in Figure 1b the forwarding path *EX* to *EX* is not excited at all (along with the possible permanent faults on that path). The signature is identical to the case in Figure 1a (assuming Performance Counters are not used) since all the instructions, even if delayed, will properly yield the correct results but using different processor paths. Even though these phenomena have

been described using as example the forwarding unit, they are applicable to all those self-test procedures that require a specific sequence of instructions to be executed without interruptions. These claims will be experimentally justified in Section IV.

III. Proposed Approach

The main intent of this paper is to propose a strategy for executing in a deterministic manner self-test routines in a multi-core context, while striving for a low system resources occupation. These requirements are those commonly found in safety-critical embedded applications, in which the software has to be predictable and the memory resources are limited.

The vast majority of computer programs exhibit the so-called *principle of locality*: that is, a given program will access a (relatively) small portion of the available address space. Two locality principles exist: temporal and spatial locality. The former states that if a given memory address is referenced, then it is likely that it will be referenced again soon. The latter stems from the observation that programs are generally executed sequentially and data are often stored in contiguous memory locations: therefore, if a given memory location is accessed, then it is likely that the locations nearby will be accessed soon. Caches leverage these principles, by storing the content the most referenced addresses (i.e., data and instructions). This provides *isolation*, considerably increasing the processor performances. Although these advantages, the caches are not deterministic since the actual increase in performance depends on the program length and organization, the cache size itself, and how often a context switch is performed. Therefore, issues could arise when using caches in conjunction with self-test procedures, since they require a precise execution.

However, it is possible to achieve a deterministic cache-based execution if the test program is executed without any interruption and it exhibits strong temporal and spatial locality. The idea is to move the self-test routine within the innermost level of caches (i.e., the ones private to each processor core), isolating its execution from the rest of the system. From the above mentioned definitions of the locality principles, it is possible to derive a general structure, that embeds the single-core version of the self-test procedure. Given a generic boot-time test program, the few modifications required are:

1) The test program should be executed twice in a loop-based fashion. The *body of the loop* (blocks *c and d* in Figure 2b) is represented by the instructions intended for testing the processor which compose the single-core self-test procedure (Figure 2a, blocks *b and c*). This allows for a strong temporal locality, since all the addresses are referenced exactly twice. During the first iteration (hereinafter *loading loop*), the test program is moved into the instruction cache. At the same time, the content of the data memory addresses referenced (if any) during this first iteration are moved within the data cache, assuming a write allocate cache memory. If this is not the case (i.e., a no-write allocate policy) each store operation must be followed by a *dummy load operation* to the same address. This will provoke a read cache miss, that in turn causes data to be moved within the data cache. Therefore, during the execution loop all the store operations will not generate a write miss, since they will find the proper data already in cache. It is important to note that during the loading loop the test program must not perform any check of the signature. Since the first execution might be still influenced by the other processors' activity, the computation of the signature is unreliable. Instead, the second iteration (the *execution loop*) is the real test program execution. Since the program is executed entirely from the caches, the signature can be computed without the risk of being influenced by the rest of the system.
2) The *entire* test procedure code must be loaded in the instruction cache during the loading loop. This feature brings spatial locality and it avoids instruction cache misses during the execution loop that could potentially alter the signature. This condition implies that:
   2.1) Conditional branches that could potentially yield a different execution flow in the execution loop must be avoided. Exceptions are those conditional branches that intentionally alter the execution flow but due to the effect of a fault. Moreover, this does not preclude the applicability of the proposed methodology to loop-based test programs, as long as by the end of the test program all the possible branches are taken.
   2.2) The size of the multi-core version (Figure 2b) of the self-test procedure must fit into the available cache memory. If the resulting test program is larger than the available cache size, it must be split into two or more smaller self-test procedures. It is important to note that this step is exclusively required if the cache memory is not large enough, and it does not compromise the fault coverage of the original single-core test procedure.
3) Both data and instruction caches should be initialized, by invalidating their content (Figure 2b, block *b*) prior the test program execution (Figure 2b, block *c and d*).

The proposed strategy based on cache memories achieves

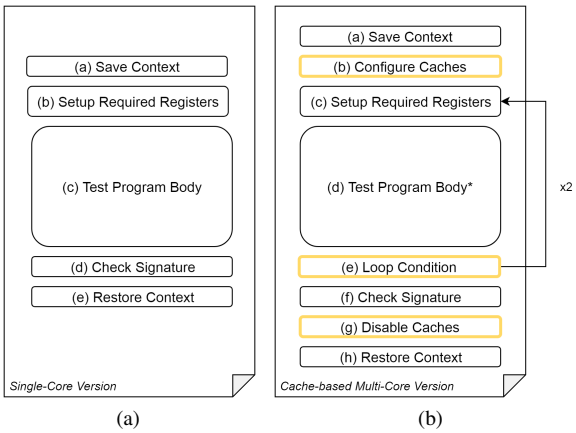


Fig. 2. The proposed Cache-based strategy. On the left-hand side the single-core version. On the right-hand side the modified multi-core test program version. In case of no-write allocate caches, the Test Program Body might be lightly modified.

the requirements of both deterministic behavior and low resources usage since:

- Caches decouple the processor from the rest of the system. Therefore, the instruction stream is not altered by other processors' activity;
- The code is allocated in the cache memories, without altering the self-test routine memory footprint.

IV. Case study and Experimental Results

This Section is organized as follows: the first subsection describes the target multi-core device. From the second subsection, the experimental results are reported. These include the evidences of issues presented in Section II and then the gathered results for the proposed methodology. Its effectiveness is also compared with other possible alternatives.

A. Case Study

The device used in the experimental part of this work was an industrial triple-core System-on-Chip, manufactured for automotive safety-critical applications ranked as ASIL D. It embeds three dual-issue processor cores. Hereinafter, these cores will be labeled as cores A, B and C. The two cores A and B are the same 32-bit processor model, while the core C is different since it implements an extended instruction set able to deal with 64-bit operands. Each processor includes two Tightly-Coupled Memories modules (for data and instructions), along with private data (4 kB) and instruction (8 kB) caches. The caches support both write allocate and no-write allocate policies (configurable before being used).

For the purpose of this work, stuck-at faults were exclusively considered. Nevertheless, the applicability of the proposed methodology is not limited exclusively to this specific fault model. The total number of stuck-at faults of these processors varies from 643,209 (core C) to 473,052 (core B). It is worth noting that although core A and B are conceptually identical, they underwent different physical design processes. Therefore, from the testing viewpoint they are quite different, since the stuck-at fault lists are different. Accordingly with the aim of this paper, the faults belonging to the Interrupt Control Unit and Hazard Detection Unit were exclusively considered. The self-test procedures developed for these units are significant examples of the complications that arise when considering a multi-core execution.

The problems related to the Hazard Detection Unit (which includes also the forwarding mechanism) were already presented in Section II. In the considered processors, the Hazard Detection Unit is composed of a Hazard Detection Control Unit and a Forwarding Logic. The former detects dependencies among issue packets, driving the forwarding paths and possibly stalls the pipeline if the forwarding is not possible. The latter is composed by the multiplexers that directly fed and collect the results produced by the different execution units of the processor. Several algorithms exist in the literature for testing these mechanisms [18], [19]: in the following it was decided to implement the one presented in [19] since it targets a multiple-issue processor. The above-mentioned testing algorithm exhaustively test all the possible existing forwarding paths, both *interpipeline* (that is, dependencies between instructions of the same issue packet) and *intrapipeline* (dependencies between instructions of two consecutive issue packets). Moreover, it leverages performance counters for tracking the number of pipeline stalls in the processor during the self-test procedure execution (for detecting wrongly inserted stalls by the hazard control unit).

TABLE I
Multi-core STLs Execution: Stalls due to the Memory Subsystem

# Active Cores	IF Stalls [clock cycles]	MEM stalls [clock cycles]
1	200,679	117,965
2	717,538	305,801
3	1,878,336	663,386

Concerning the Interrupt Control Unit, *synchronous imprecise interrupts* were examined. Such class of interrupts are still generated as consequence of a particular instruction being executed (i.e., synchronously) and from sources within the CPU. But, unlike *precise interrupts* [20], the imprecise ones are not recognized immediately, but only after that a variable number of instructions are executed beyond the interrupting instruction. The actual number of instructions depends on the instructions stream entering the pipeline, which is highly unpredictable in a multi-core system. Therefore, also the self-test procedures targeting these interrupts suffer of an unstable signature that varies depending on the other processors' activity. For testing this mechanism, a self-test procedure based on the strategy presented in [21] was implemented. The second column of Table II and the third of Table III report the number of faults within these units. Finally, caches were configured with a write allocate policy: therefore, for both test programs, it was not required to insert additional load operations to avoid write misses in the execution loop (as explained in Section III). Furthermore, for both test programs, it was not necessary to split them, since the instruction cache was large enough to contain the entire self-test procedure code.

B. Uncertainties in multi-core SoC

A first set of experiments consisted in analysing the behavior of the STL in a multi-core context. Considering the system under analysis, two STLs were developed (core A and B share the same STL). The test programs targeting imprecise interrupts and hazard detection unit were not included in the library for this initial set of experiments, since their behavior was analyzed separately. The STLs were executed in parallel on the physical microcontroller, with a software structure similar to the one presented by the authors of [13]. Their execution was tracked leveraging an external debugger, that monitored the number of clock cycles of stall due to the memory subsystem in each processor core. Table I reports the gathered measurements. As it can be noticed, when moving from a single-core scenario (in which all the other cores are completely turned off) to a triple-core scenario, the number of stalls in the system increased considerably. As it can be noted,

the major source of stalls is the instruction fetch unit (second column of Table I). This is a direct consequence of the higher bus contention: the instruction fetch operations are delayed due to the other processors requests, and as a consequence the pipeline is stalled. Moreover, it is worth noting that the figures in the second and third row of Table I represent *average values* gathered across several executions. The actual number of clock cycles of stall varies depending on the initial SoC configuration (and therefore it is not predictable).

C. Uncertain Fault Coverage

From the experiments described above it is clear that the behavior of an STL is highly unpredictable in a multi-core context, since it is influenced by the whole system activity. The second set of experiments focused on demonstrating the effects of these pipeline stalls on the self-test procedures. Specifically, these experiments involved the achievable fault coverage on the processor hazard detection unit. For these experiments, the SoC post-layout gate-level netlist and a commercial fault simulator were used. As extensively explained in Section II and demonstrated with the previous experiments (Table I) Performance Counters (PCs) are unreliable in a multi-core scenario. Therefore, when they contribute to the self-test procedure signature, a straightforward solution for dealing with the instability of the signature might be removing the usage of PCs, sacrificing fault coverage. However, as depicted in Table II this is not enough for guaranteeing a *deterministic fault coverage* of the forwarding logic. First, the algorithm [19] was modified, removing the usage of PCs. Then, the obtained self-test procedure was executed in parallel on the different processors considering different scenarios: number of active cores (two or three), code position in memory (low, mid and high Flash addresses) and different code alignment options (e.g., aligned at word, double-word or double double-word).

TABLE II
FORWARDING LOGIC FAULT SIMULATION RESULTS

Core	# of Faults	min - max FC [%] no caches no PCs	FC [%] with caches no PCs
A	53,298	64.14 - 75.19	79.61
B	57,506	63.61 - 79.59	82.08
C	113,212	56.24 - 66.48	68.79

Each of these logic simulations was then fault simulated, and the results are shown in third column of Table II. For sake of conciseness, the minimum and the maximum value of fault coverage are reported only. As it can be observed, the fault coverage considerably oscillates: in the worst case, it was observed a difference of about 16%. It is important to note that the signature did not change during the logic simulations *and yet the fault coverage varied significantly*. These fluctuations depend on how many issue packets consecutively (namely in consecutive clock cycles, without any stall in between) enter the processor pipeline, activating different forwarding paths. On the contrary, when executing the self-test procedure embedded in proposed cache-based approach (fourth column of Table II), the fault coverage significantly increased (about the 4% in the best case) while being stable across the different scenarios. The fault coverage obtained for core C is lower compared to the one of cores A and B because the multiplexers are 64-bit wide to support 64-bit operations. However, general purpose registers are still 32-bit wide. Therefore, the signature must be represented using 32 bits, which causes some faults effects to be masked. Nevertheless, the reader should note that improvements of the already existing algorithm for the forwarding logic would have been outside the scope of this work. For this reason, increasing further the fault coverage was not considered.

TABLE III
ICU AND HDCU FAULT SIMULATION RESULTS

Core	Module	# of Faults	FC Sigle-Core no caches [%]	FC Multi-Core with caches [%]
A	ICU	14,230	46.57	51.36
	HDCU	16,096	62.53	70.37
B	ICU	13,149	46.39	50.97
	HDCU	15,783	63.84	70.12
C	ICU	13,888	54.94	60.91
	HDCU	19,931	65.66	68.09

D. Unstable Signature

The third set of experiments (Table III) concerned Interrupt Control Unit and Hazard Detection Control Unit (ICU and HDCU respectively). For the HDCU, the complete algorithm of [19] was used (namely with performance counters). For the ICU, the aforementioned self-test procedure based on [21] was used. The fourth column of Table III represents the fault coverage figures when the self-test procedures were executed in the selected SoC in a single-core scenario (i.e., with the other cores switched off) without resorting to the proposed approach. In this scenario, the signatures produced by the test programs were stable as the fault coverage. However, when moving to a multi-core execution *without using the caches* the test procedures inevitably failed in any configuration. Introducing caches, the produced signatures become stable, and therefore the fault coverage can be computed. As the reader can notice, the achieved fault coverage in a multi-core execution is higher than in the single-core scenario. This lower fault coverage stems from the fact that the memory subsystem introduces 8 clock cycles of latency when fetching an issue packet from the Flash even in a single-core execution. Thus, it is not possible to fully excite all the forwarding paths or trigger correctly all the imprecise interrupts. Furthermore, while for the HDCU the coverage was similar over the three processors, the coverage for the ICU is about 10% higher in the core C. This arises from the implementation of the ICU itself. In details, the unit exposes some software-accessible registers for differentiating among the possible sources of interrupt. In the core A and B, different interrupt events are mapped to the same bits. As a result, even here some fault effects are masked (unlike core C).

E. Comparisons with other solutions

Finally, a common alternative to the proposed one consists in exploiting the processor Tightly-Coupled Memories (TCMs, also known as scratchpad memories) [22], [23]. This approach

TABLE IV
TCM-BASED VERSUS CACHE-BASED APPROACHES FOR IMPRECISE INTERRUPTS

Approach	Overall Memory Overhead [bytes]	Execution Time [clock cycles]
TCM-based	2,874	16,463
Cache-based	0	18,043

is typically adopted for the execution of real-time programs. Such programs are copied (during the system boot) and then executed from the instruction TCM when required. Conceptually, TCMs are similar to caches since they consist in a bank of SRAM local to each processor. Unlike caches, there is not the concept of cache miss or hit, since data or instructions have to be copied explicitly to these memories before being used. This approach shows most of the advantages of the proposed one. However, the fundamental drawback is that part of the TCM should be *permanently reserved* for test purposes (the amount of extra memory occupied is proportional to the size of the test program). Clearly, this impacts negatively on both portability and flexibility of the STL. Table IV compares the two strategies (namely TCM-based versus cache-based execution) for the self-test procedure targeting the imprecise interrupts (similar results were obtained also for the hazard detection units, but they were not reported for conciseness). Since both approaches require few additional instructions to be implemented, the flash overhead is negligible and not reported. *The same reasoning applies also for the fault coverage, which is the same for both*. Concerning the TCM-based, the execution time consists in the time required for copying the entire self-test procedure in the Instruction TCM and then execute from there the self-test program. For the cache-based one, it is the time required for executing as in Figure 2b. As it can be viewed, *the cache-based approach does not increase the overall memory footprint of the self-test procedure*, while it requires to be executed slightly more than 1,500 clock cycles compared to the TCM-based approach. It is worth noting that this overhead might be negligible when the STL is executed at-speed (8.25μs when the considered SoC operates at its maximum frequency of 180 MHz).

V. CONCLUSION

This paper described a cache-based approach for achieving a deterministic execution of self-test procedures when deployed in field in a multi-core SoC. In this context, the suggested methodology is able to deliver stable signature and deterministic fault coverage, *without requiring additional on-chip resources*. Through the experiments, it was demonstrated the applicability to any self-test procedure without altering its overall memory footprint. On the other hand, it requires slightly more clock cycle to be executed compared to other strategies (e.g., the TCM-based ones). While considering stuck-at faults, few specific test programs exhibit these issues in a multi-core execution. Instead, it might be further emphasized with delay faults which require test patterns applied in a timed sequence.

REFERENCES

[1] F. Reimann *et al.*, "Advanced diagnosis: Sbst and bist integration in automotive e/e architectures," in *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2014, pp. 1–6.

[2] (2019) ARM Software Test Library: [Online]. Available: https://www.arm.com/products/development-tools/embedded-and-software/software-test-libraries.

[3] (2019) Infineon Software Test Library: [Online]. Available: https://www.hitex.com/tools-components/software-components/selftest-libraries-safety-libs/pro-sil-safetcore-safetlib/.

[4] (2019) Cypress Software Test Library: [Online]. Available: http://www.cypress.com/file/249196/download.

[5] (2019) Renesas Software Test Library: [Online]. Available: https://www.renesas.com/en-eu/products/synergy/software/add-ons.html#read.

[6] (2019) Microchip Software Test Library: [Online]. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/52076a.pdf.

[7] P. Bernardi *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.

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

[9] A. Paschalis *et al.*, "Deterministic software-based self-testing of embedded processor cores," in *Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001*, March 2001, pp. 92–96.

[10] M. Psarakis, D. Gizopoulos, E. Sanchez, and M. S. Reorda, "Microprocessor software-based self-testing," *IEEE Design Test of Computers*, vol. 27, no. 3, pp. 4–19, May 2010.

[11] N. Kranitis *et al.*, "Hybrid-sbst methodology for efficient testing of processor cores," *IEEE Design Test of Computers*, vol. 25, no. 1, pp. 64–75, Jan 2008.

[12] L. Chen and S. Dey, "Software-based self-testing methodology for processor cores," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 3, pp. 369–380, March 2001.

[13] A. Floridia *et al.*, "A decentralized scheduler for on-line self-test routines in multi-core automotive system-on-chips," in *2019 50th International Test Conference*, Nov 2019, pp. 1-10.

[14] M. Lv, W. Yi, N. Guan, and G. Yu, "Combining abstract interpretation with model checking for timing analysis of multicore software," in *2010 31st IEEE Real-Time Systems Symposium*, Nov 2010, pp. 339–349.

[15] A. Apostolakis, D. Gizopoulos, M. Psarakis, and A. Paschalis, "Software-based self-testing of symmetric shared-memory multiprocessors," *IEEE Transactions on Computers*, vol. 58, no. 12, pp. 1682–1694, Dec 2009.

[16] M. A. Skitsas, C. A. Nicopoulos, and M. K. Michael, "Daemonguard: Enabling o/s-orchestrated fine-grained software-based selective-testing in multi-/many-core microprocessors," *IEEE Transactions on Computers*, vol. 65, no. 5, pp. 1453–1466, May 2016.

[17] T. Hsieh *et al.*, "Tolerance of performance degrading faults for effective yield improvement," in *2009 International Test Conference*, Nov 2009, pp. 1–10.

[18] M. Psarakis *et al.*, "Systematic software-based self-test for pipelined processors," in *2006 43rd ACM/IEEE Design Automation Conference*, July 2006, pp. 393–398.

[19] P. Bernardi *et al.*, "Software-based self-test techniques for dual-issue embedded processors," *IEEE Transactions on Emerging Topics in Computing*, pp. 1–1, 2018.

[20] J. E. Smith and A. R. Pleszkun, "Implementing precise interrupts in pipelined processors," *IEEE Transactions on Computers*, vol. 37, no. 5, pp. 562–573, May 1988.

[21] P. Singh, D. L. Landis, and V. Narayanan, "Test generation for precise interrupts on out-of-order microprocessors," in *2009 10th International Workshop on Microprocessor Test and Verification*, Dec 2009, pp. 79–82.

[22] J. Ax *et al.*, "Coreva-mpsoc: A many-core architecture with tightly coupled shared and local data memories," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 5, pp. 1030–1043, May 2018.

[23] R. Banakar *et al.*, "Scratchpad memory: a design alternative for cache on-chip memory in embedded systems," in *Proceedings of the Tenth International Symposium on Hardware/Software Codesign. CODES 2002 (IEEE Cat. No.02TH8627)*, May 2002, pp. 73–78.