A Decentralized Scheduler for On-line Self-test Routines in Multi-core Automotive System-on-Chips

A Decentralized Scheduler for On-line Self-test Routines in Multi-core Automotive System-on-Chips

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

***Abstract*—Modern System-on-Chips (SoCs) deployed for safety-critical applications typically embed one or more processing cores along with a variable number of peripherals. The compliance of such designs with functional safety standards is achieved by a combination of different techniques based on hardware redundancy and in-field test mechanisms. Among these, Software Test Libraries (STLs) are rapidly becoming adopted for testing the CPU and peripherals modules. The STL is usually composed of two sets of self-test procedures: boot-time and run-time tests. The former set is typically executed during the boot or power-on phase of the SoC since it requires full access to the available hardware (e.g., these programs need to manipulate the Interrupt Vector Table and to access the system RAM). The latter set instead, is designed to coexist with the user application and can be executed without requiring special constraints. When the STL is intended for testing the different cores within a multi-core SoC, the concurrent execution of the boot-time self-tests becomes an issue since this could lead to a longer power-up phase and excessive utilization of system resources. The main intent of this work is to present the architecture of a decentralized software scheduler, conceived for the concurrent execution of the STL on the available cores. The proposed solution considers the typical constraints of an STL in a multi-core scenario when deployed in field, namely minimum system resources usage (i.e., code and data memory). The effectiveness of the proposed scheduler was experimentally evaluated on an industrial STL developed for a multi-core SoC manufactured by STMicroelectronics.**

I. Introduction

In the last decade the complexity of electronics devices deployed for safety-critical applications has been growing exponentially. Multi-core architectures are the answer of the semiconductor manufacturers for satisfying the performance requirements whilst being limited by the maximum power consumption of embedded applications. In the context of the automotive domain, the ISO 26262 standard regulates the usage of electronic devices and imposes strict reliability figures. From an industrial perspective, different solutions (commonly referred also as safety mechanisms) are together adopted for achieving the highest safety level (e.g., ASIL C or D) [1]. Such safety mechanisms fall in two main categories: hardware-based and software-based safety mechanisms. The former ones can be further distinguished depending on the functionality they are intended to perform:

- Fault detection and/or correction mechanisms: Triple Modular Redundancy (TMR), Lockstep Computing, End-to-End Error Correction Code for memories [2], [3].
- In-field testing mechanisms: Logic and Memory Built-in Self-Test (LBIST and MBIST, respectively).

Several examples of both LBIST and MBIST exist [4]–[8], and when correctly adopted they yield high fault coverage metrics. Typically, they are applied during the so-called Power-on Self-Test (POST), that is the in-field test performed after the device is powered up. Nevertheless, functional safety standards also impose testing the device periodically at run-time, that is when the system is already executing its functionalities (e.g., executing the application software). This type of test is normally referred to as on-line testing.

Software-based mechanisms better fit the on-line testing requirements. Normally, they consist in a set of software procedures or test programs composing a Software Test Library (STL). These programs are executed by the processor core and their main target is to test the processor core and eventually the peripherals surrounding it. This strategy, initially proposed by [9], has been studied by different research groups [10]–[15], and later extended targeting safety-critical devices used in the automotive field [16], [17]. These self-test procedures are usually developed by the semiconductor company, which owns all the structural information about the device and can guarantee that their execution achieves an adequate figure in terms of fault coverage. Once available, the self-test procedures are then integrated by the system company (often corresponding to a Tier1 company in the automotive domain) into the application software and invoked when required (e.g., during the power-on, periodically, or during the application idle times). In this way, the required safety level can be obtained without transferring any sensible information from the semiconductor to the system company. Furthermore, industrial STLs are designed to be configurable: the final customer (i.e., the system company) decides whether some self-test routines are required or not. As an example, if the floating-point unit is not used, the customer may decide to not execute the self-test procedures targeting that unit. This approach is today widely supported by many semiconductor

and IP companies, such as STMicroelectronics, Infineon [18], Cypress [19], Renesas [20], Microchip [21], ARM [22].

Concerning the automotive domain, the self-test procedures can be distinguished [16], [22] in:

- *Boot-time self-test routines*: they are executed by the target processor core during the boot phase of the SoC, when the system is entering the so called on-line phase. Tests belonging to this category modify the processor Special Purpose Registers, the Interrupt Vector Table, trigger exceptions and preemption is not allowed. Moreover, for the sake of test purposes, they require to access specific addresses in the shared portion of the system RAM, *outside the boundaries of the processor stack frame*.
- *Run-time self-test routines*: they are conceived to coexist with the application source code. Generally, they target mainly the computational units (e.g., arithmetic units) within the processor core. They are executed in real-time, they do not alter the processor status and can be interrupted in case high priority tasks require to be executed. Differently than boot-time tests, run-time ones do not access system RAM addresses outside the processor stack frame.

Considering single-core devices, a more comprehensive description of these tests and guidelines for an efficient development can be found in [16]. However, in that paper, the authors did not addressed the issues related to the usage of an STL in a multi-core context. The most relevant constraints of an STL oriented to test on-line a multi-core system are described below:

- *Minimum memory resources usage*: this stems from the fact that the STL source code normally coexists with an operating system or more in general with an application software. Since in embedded systems the memory resources are limited, the STL should interfere minimally with the user application. This requirement implies a limited use of memory resources to allocate the STL source code and also a reduced number of system RAM portions to be reserved for test purposes.
- *Avoid any source of non-determinism*: some test programs require proper sequence of instructions to be executed, that might be modified by the processor speculative units, such as the Branch Predictors and Caches. For the sake of determinism (at least for boot-time tests), these speculative modules are normally disabled and eventually enabled only when needed by specific test programs.
- *Whenever possible, reduce conflicts due to shared resources usage*, such as Interrupt controllers and other peripherals.

II. Related Work

A. Motivations

To the best of our knowledge, the vast majority of the available Software-Based Self-Test (SBST) strategies [12] were devised considering only a single processor core at a time. When the STL has to be executed in field on the different processor cores of a multi-core system, a possible solution is to serially test one processor core at a time. Although straightforward, potential problems could arise when dealing with the boot-time tests. Despite the fact that the real-time constraints are more relaxed compared to the run-time routines, the boot-time routines are executed during the namesake phase of the System-on-Chip after it is turned on. Therefore, the serial execution could delay excessively the user application. For speeding-up the test application time, a better solution could be to execute the STL in parallel among the different cores. Clearly, the parallel execution of the entire STL is not always feasible since often some boot-time test programs use shared resources, such as the system RAM. The shared memory subsystem may represent a bottleneck for the concurrent execution of the test, due to the access contention among the different processor cores. Hence, the embedded software running in a multi-core scenario suffers from a limited timing predictability [23]. This issue is further emphasized from the fact that, unlike the traditional multi-core applications, in most of the cases there exists a unique copy of the STL in memory accessed by all the processor cores. Therefore, a suitable software scheduler is needed for managing the shared resources accessed by the different test programs (e.g., the system RAM).

B. State of the Art

Regarding the state-of-the-art self-test routines scheduling, the authors of [24] presented a test selection algorithm oriented to the minimization of the execution time under real-time constraints. The same topic was also addressed in a detailed study presented in [25]: the paper presents some scheduling alternatives when the self-test procedures coexist with hard real-time tasks. By an appropriate test selection, it was shown how the timing deadline can be accomplished without sacrificing the self-test quality (i.e., the fault coverage). Similar analyses but in a multi-core scenario were presented in [26] and [27]. In the former, the impact of a periodic testing on system availability was assessed. Using a suitable exploration framework, it was shown how to derive scheduling policy such that the test execution time is minimized and consequently maximizing the system availability. In the latter instead, a power-aware periodic scheduling is proposed with minor performance penalty while satisfying the imposed power consumption constraints.

The most closely relevant works to the one presented in this paper, albeit in a quite different testing scenario, are [28]–[30]. In [28] a scheduling algorithm of self-test routines for shared-memory multi-processor system is proposed. The algorithm is based on an optimized usage of system caches and code allocation in order to minimize the latency introduced by the memory subsystem. Instead in [29], it is shown how to exploit thread-level parallelism to improve the execution of self-test routines in each core of a multiprocessor chip. The same research group then presented in [30] an effective strategy for multi-threaded multi-core systems oriented at maximizing the

execution parallelism of the self-test routines without affecting the fault coverage.

However, the aforementioned papers dealt with the self-test procedures for end-of-manufacturing testing. Such test programs have fewer constraints in terms of memory usage and resources availability than the ones intended for in-field testing. Moreover, the approaches use caches extensively, which are not applicable in the scenario targeted in this work.

C. Contributions

All the above-mentioned studies treat either multi-core end-of-manufacturing testing or run-time on-line testing. Conversely, a first analysis of boot-time self-test procedures execution in a multi-core scenario is presented in [31]. With respect to [31], the aim of this work is to propose a *decentralized scheduler* for boot-time self-test procedures that:

1) *maximizes the concurrency* of the tests among the different cores while *minimizing the system resources usage*;
2) can be used in compliance with the *actual industrial STL development methodology*;
3) *maintains the same fault coverage* of the STL as when executed in a single-core scenario.

The rest of the paper is organized as follows: Section 3 introduces some practical concepts of the STL development flow that constraints the concurrent execution in the multi-core scenario along with the concepts about software synchronization in the multi-processor scenario. In Section 4 the proposed scheduler is described in details. The experimental validation of the scheduler is discussed in Section 5. Finally, Section 6 concludes the paper summarizing the obtained results and future directions.

III. Background

The purpose of this Section is to discuss the most relevant constraints for the concurrent execution of an STL and briefly overview the most common techniques for synchronizing software executed by different processor cores.

A. Constrains for the concurrent execution of a STL

Typically, some boot-time self-test procedures exploit specific portions of the system RAM for improving the test of the targeted modules. For avoiding *uncertain fault coverage*, the addresses are fixed and chosen outside the boundary of the processor stack frame. As an example, test programs targeting the effective address calculation mechanisms extensively use strategies based on fixed addresses [32]. This approach is also adopted when the targeted module requires the source code being executed from specific memory locations. These test programs are normally copied to and executed from the system RAM, since reserving fixed addresses in the code memory would be too restrictive and negatively impact the portability of the STL. When the STL is executed in a multi-core System-on-Chip, those memory addresses should belong to the shared region of the RAM (i.e., a region allocated for common data between processors). Since in real applications, it is not feasible reserving multiple portions of this shared region exclusively for the STL, it is evident that a set of the boot-time self-test procedures cannot be executed in parallel due to conflict in accessing the shared portion of the RAM. Indeed, if multiple test programs access simultaneously the same shared region, the outcome is unpredictable.

B. Multi-processor software synchronization

Semaphores are one of the existing methods for achieving synchronization of software executed on different processors. A semaphore is an abstract data type (in most of the cases a shared variable) that regulates the access to a common resource by multiple processors. Depending on the number of processors allowed to access the common resource, the semaphores are distinguished in counting and binary semaphores. The latter are also called *mutex*, since exclusively one processor at a time can access the shared resource. When the processor is accessing the shared resource is said to be executing the code of the *critical section*. The basic idea is that the accesses to a shared resource are guarded by a suitable semaphore. Before entering the critical section, the processor checks the status of the semaphore. If the semaphore is available (i.e., *unlocked*), the processor tries to *acquire* the semaphore, locking the access to the shared resource. Independently from the low-level implementation, the mechanism used for checking and then acquiring the semaphore must guarantee that the two operations are executed atomically. That is, they appear to be indivisible from the other processors perspective. Finally, before leaving the critical section, the semaphore is *released* unlocking the access to the shared resource. Depending on the considered Instruction Set Architectures (e.g., ARM, MIPS, PowerPC, RISC-V), there exist different atomic instructions that perform a read-modify-write operation on a given memory location.

IV. Proposed Approach

The end goal of this paper is to propose a decentralized software scheduler intended for the concurrent execution of boot-time self-test routines in a multi-core SoC. The proposed scheduler considers the typical constraints of an STL when deployed in field, namely minimum system resources usage and compliance with the actual industrial STL development methodology, while maximizing the concurrency. These requirements imply:

- a unique copy of the STL in the code memory and portion of system RAM available for the test;
- unaltered fault coverage with respect to a single-core scenario;
- deterministic execution time.

A. Problem Formulation

The reader should note that the problem addressed in this paper is quite different from a generic multi-core real-time scheduler for embedded systems. Normally, given a set of tasks $\Lambda = \{\lambda_0, \ldots, \lambda_m\}$ to be executed, and a set of processing units $P = \{p_0, \ldots, p_n\}$, the goal of a scheduler is to assign to each processing unit $p_j \in P$ a set of tasks $\Gamma \subseteq \Lambda$ so

that the overall execution time is minimized. When dealing with a STL, Λ corresponds to the set of self-test procedures composing the STL while P represents the different cores to be tested. The fundamental difference relies on the fact that each $\lambda_i \in \Lambda$ must be executed on each core $p_j \in P$. Therefore, in this scenario the scheduler should guarantee the execution of all the self-test procedures while avoiding conflicts due to common shared resources (e.g., the system RAM), so that the overall test execution time is minimized.

Two possible approaches exist to schedule the accesses to a shared resource in multi-processors environments: centralized and decentralized. In the former, there is a high degree of control over the scheduling process, since all the requests are processed by a unique scheduler. Although it is conceptually easier to reason and design according to this paradigm, the drawback is that the scheduling is not so efficient in terms of performances. Instead, decentralized schedulers represent a more efficient solution, since they have an intrinsic distributed nature which takes full advantage of the underlying multi-processors system. Typically, they are built upon a first come, first served policy. Therefore, there is less control over the scheduling itself. The proposed scheduler falls into this category: each processor executes its own local software scheduler, interacting each others through a synchronization mechanism based on *mutex*.

B. The Decentralized Scheduler

Each local scheduler (described in Algorithm 1) concurrently executes all the *Test Programs* (*TPs*) included in an ordered set, hereinafter called *TestTable*, which defines the execution order of the self-test procedures composing the STL. It is important to note that the order specified in the *TestTable* is identical for each processor. The *TPs* that cannot be executed in parallel due to conflicting accesses to a shared resource (e.g., the system RAM) are also present in a second set called the *ShareResource* set. The relation between these two sets is *ShareResource* $\subseteq$ *TestTable*. In the following, it is assumed that *TestTable* is ordered so that the test programs in *ShareResource* are the first ones in the *TestTable* sequence. The actual number of *TPs* executed is tracked by a different set, the *PendingList*. Differently than in *TestTable*, the order of the elements composing *PendingList* and *ShareResource* is irrelevant.

At the beginning, |*PendingList*| = |*TestTable*|. The scheduler sequentially selects one *TP* at a time from *TestTable* (coherently with the specified order, line 5), and it checks whether the selected *TP* is still present in *PendingList* (that is, not yet executed, see line 6). If so and at the same time the selected *TP* does not belong to the *ShareResource* set, the *TP* can be executed. Once executed, it is also removed from *PendingList* (line 23), reducing its cardinality. Contrarily, if the *TP* is also present in *ShareResource* (line 7), the *TP* can be executed given that the common resource is not busy. This is achieved through a suitable mutex (line 8), which signals to each local scheduler whether another scheduler is currently executing a *TP* in the *ShareResource* set. Therefore, the critical section corresponds to the portion of code executing tests belonging to *ShareResource* (lines 8 to 16). It is important to note that a mutex is required since only one local scheduler at a time can access the critical section. If the resource is not free, the *TP* is skipped. Instead of performing busy waiting (i.e., waiting for the resource to be freed), the scheduler tries to execute another *TP* (line 19). If the resource is free, the *TP* is executed (line 9) and removed from *PendingList* (line 10). The STL execution completes when there are no *TPs* to be executed, that is *PendingList* is empty (line 3).

As it can be noticed, the scheduler does not release the mutex (lines 11 to 16) as long as the next sequential *TP* still uses the shared resource. For this reason, the proposed scheduler is said to be *selfish*: the first local scheduler that acquires the mutex, executes consecutively all the *TPs* within the *ShareResource* set without freeing the shared resource. As it is experimentally proved in the experimental part of this work, not releasing the mutex along with the devised *TestTable* ordering assumes a crucial aspect for the determinism and the efficiency of the scheduler.

Figure 1 shows a practical example of the scheduler described above. For the sake of a better understating, let us consider the simplified case in which two processor cores

Algorithm 1: Selfish Decentralized Scheduler Algorithm

```
1  PendingList ← TestTable;
2  MaintainMutex ← false;
3  while |PendingList| ≠ |∅| do
4      i ← 0;
5      for TP_i ∈ TestTable do
6          if TP_i ∈ PendingList then
7              if TP_i ∈ ShareResource then
8                  if Acquire(Mutex) is successful ∨
                    MaintainMutex is true then
9                      Execute TP_i;
10                     PendingList ← PendingList\{TP_i};
11                     i ← i +1;
12                     if TP_i ∈ ShareResource then
13                         MaintainMutex ← true;
14                     else
15                         MaintainMutex ← false;
16                         Release(Mutex);
17                     end
18                 else
19                     i ← i +1;
20                 end
21             else
22                 Execute TP_i;
23                 PendingList ← PendingList\{TP_i};
24                 i ← i +1;
25             end
26         end
27     end
28 end
```

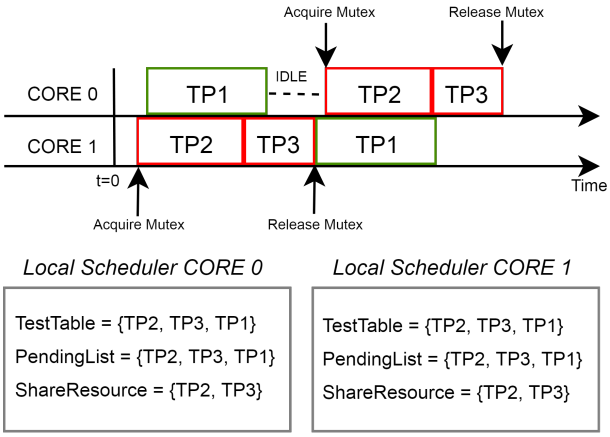

Fig. 1. Timeline of the proposed scheduler.

(CORE 0 and CORE 1) must execute a boot-time STL composed of three self-test procedures (TP1, TP2 and TP3). Two of these, TP2 and TP3, require the usage of a shared portion of the RAM. It should be noticed that TP2 and TP3 are scheduled at the beginning of the *TestTable* sets. Initially, as represented in the figure, the *PendingList* sets for both processors are equal to *TestTable*. Therefore, with the same formalism used in this Section, for each core the three sets are composed as depicted in Figure 1.

Initially, the mutex is free. Assuming CORE 1 is the first one that acquires the mutex, it starts executing TP2 and TP3 (being the firsts in *TestTable*) without releasing the mutex. At the same time, CORE 0 checks the status of the mutex, which is locked by CORE 1. Thus, it cannot execute any self-test procedure in the *ShareResource* set and it executes TP1 (being the first self-test procedure not requiring a common resource). Once CORE 0 terminates the execution of TP1 (it is removed from CORE 0 *PendingList*), CORE1 has not yet freed the mutex. Therefore, CORE0 waits until the resource is freed. Once TP3 terminates the execution on CORE 1, the mutex is released. At this point CORE 1 has to execute TP1 (the last program in CORE 1 *PendingList*), while CORE 0 can acquire the mutex and execute the remaining self-test procedures in its *PendingList* (i.e., TP2 and TP3).

By adopting the proposed scheduler described beforehand, the concurrency of the STL execution is maximized since while the shared resource is busy the local schedulers execute other self-test procedures. This does not impact the fault coverage, since the test programs are not preempted and thus the instructions flow of a given test program is not altered. Although there is a local scheduler for each processor core, the copy in the code memory is unique and shared by each processor. Therefore, the resources utilization is minimal. In the next Section, experimental evidences are provided supporting these claims.

V. Case Study and Experimental Results

This Section is organized as follows: the first subsection describes the industrial case study used for the experimental validation of the proposed scheduler. Then, for the sake of comparison, in the second and third subsections the obtained performances (in terms of execution time) and limitations of different scheduling alternatives are reported: unlike the proposed method, a serial scheduler and several non-selfish decentralized schedulers are analyzed. Finally, in the last subsection the same analyses are also performed for the proposed scheduler and compared with the previous ones.

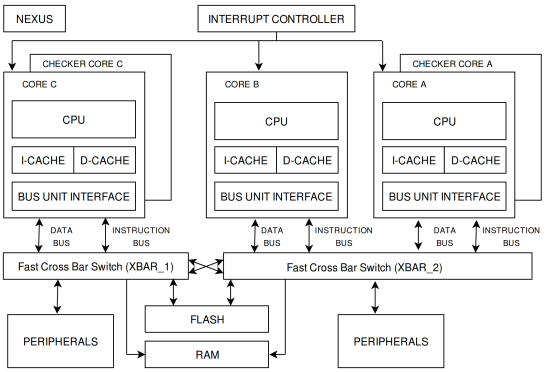

Fig. 2. SPC58NN84 internal architecture.

A. Case Study and Experimental Setup

The device used to prove experimentally the effectiveness of the proposed decentralized scheduler was the SPC58NN84, a triple-core SoC manufactured by STMicroelectronics, intended for automotive ASIL D applications. The SoC embeds three 32-bit dual-issue z4256n3 PowerPC-compliant processors. Each processor core includes a local SRAM for code and data, along with private data and instruction caches. For the sake of generality and experiments reproducibility, the local SRAM memories were not used (since this feature is not always available among the different automotive SoCs). Also the caches were not used, because they might compromise the outcome of some self-test procedures (also in terms of fault coverage) due to their non-deterministic behavior. Additional safety mechanisms are available in the considered SoC: two of the three processors are paired with an additional checker core in a delayed lockstep configuration. Additionally, the memories are equipped with ECC. The architecture of the SoC is depicted in Figure 2. The SoC includes 6Mbytes of Flash, 128Kbytes of system RAM and different peripherals shared among the three cores.

Concerning the STL, it comprises 104 self-test procedures: 34 of these are run-time tests, while the remaining 70 are boot-time tests. 13 out of 70 use a shared region of the system RAM. Therefore, they belong to the *ShareResource* set and cannot be executed concurrently. The memory footprint of the single-core STL is 429Kbytes. It is common practice to execute the run-time tests along with the boot-time ones during the device startup, to obtain the highest possible fault coverage. Then, the run-time test programs are executed periodically when the system is on-line. Therefore, for reproducing a realistic case study in the performed experiments, all the 104 self-test procedures are executed during the device startup. Finally,

the STL covers the 84.41% of the possible 729,522 stuck-at faults affecting the processor core. Generally, a stuck-at fault coverage grater than the 80% is considered acceptable for the processor core. This value stems from functional safety analyses performed by Functional Safety engineers, considering the number of faults within the core with respect to the total number of faults of the entire SoC. The library is executed in a single-core scenario in about $29ms$ with a system clock frequency of 16MHz. This execution time includes an initialization phase and a test phase. During the former the processor internal state (namely the Special Purpose Registers) is set along with specific Interrupt Service Routines for handling exceptions generated by test programs and other utility functions for sake of testing. The latter is the phase in which the test programs are actually executed.

The measurements of the different execution times were collected directly on the physical device, leveraging the on-chip hardware timers. For all the performed experiments the system clock frequency was the default one (i.e., 16MHz). Increasing the operating frequency was not considered as a solution since it would be out of the scope of this work. Furthermore, when referring to multi-core decentralized schedulers, the following setup was used:

- to reproduce the worst possible scenario, the measurements were gathered aligning the execution of the processor cores;
- when the number of active cores is two, the third core is switched off completely for avoiding influencing the outcome of the measurement;
- the hardware timer was started once the processors were aligned and then stopped once the last processor completed the test;
- all the performed measurements include the aforementioned initialization (being local to each core) and the test phase.

The considered device offers two main solutions for implementing the synchronization mechanism described in Section 2: hardware semaphores and decorated instructions. The former is a peripheral embedded in the SoC that implements directly in hardware the functionalities of a semaphore. The decorated instructions on the other hand are PowerPC-specific store/load instructions that implement an atomic read-modify-write mechanism on memory locations. In the following, all the discussed implementations are based on the decorated instructions, since they yield higher performances compared to hardware semaphores. Indeed, the average time for accessing a hardware semaphore (i.e., a peripheral) is about 190 clock cycles, while the decorated instructions require considerably fewer clock cycles (17 only).

B. Analysis of serial scheduling

A preliminary set of experiments consisted in measuring the execution time of the STL when using a serial scheduler (i.e., the STL is executed serially on each core). These measurements are used here as a reference point being the upper bound of any multi-core scheduler. The measurements are shown in Table I and were gathered considering the STL executed on 2 and 3 cores.

TABLE I
PERFORMANCES OF THE SERIAL SCHEDULER @16MHZ

# Active Cores	Execution Time [ms]
2	57.15
3	88.01

TABLE II
PERFORMANCES OF DIFFERENT DECENTRALIZED SCHEDULERS @16MHZ

Decentralized Scheduler	Execution Time [ms]	
	2 Active Cores	3 Active Cores
DS1	40.18	71.93
DS2	49.31	79.89
DS3	39.87	66.80
DS4	41.12	71.42
DS5	41.81	79.80

C. Analysis of Non-selfish decentralized schedulers

To prove that the proposed decentralized scheduler is valid, a second set of experiments focused on analyzing the performances of different decentralized scheduling algorithms. They differ from the proposed one since they cannot be considered selfish (that is, the mutex is released even though the next sequential self-test procedures still belong to the *ShareResource* set). The considered schedulers are listed in Table II.

Each of these decentralized schedulers considers different formats of *TestTable*: in *DS1* the order of the self-test procedures is random. A total of 30 random orders were generated, and *DS1* represents the best random ordering. It is worth to underline that the differences between the generated random orders were minimal. Therefore, for sake of conciseness, exclusively 30 random orders were generated and the best order is reported. *DS2* orders the self-test procedures so that those included in *ShareResource* are executed first. The order of the self-test procedures within *ShareResource* is random. Differently, in *DS3* the self-test procedures within *ShareResource* are ordered according to the duration of the self-test procedures themselves (with a descending order). For both *DS2* and *DS3* the remaining self-test procedures (i.e., those not included in *ShareResource*) are ordered randomly. Finally, *DS4* and *DS5* considers the self-test procedures still ordered according to their duration (descending and ascending order respectively), but independently from the fact that they could also belong to the *ShareResource* set. As it can be viewed in Table II, *DS3* is the decentralized scheduler yielding the best performances. This is justified by the fact that it is the scheduler that better reduces the memory access contention, as shown in Table III.

Table III reports the values of the on-chip performance counters for the triple-core scenario (being the worst case from the access contention viewpoint). The first column reports the schedulers names. In the second column, the number of clock cycles corresponding to stalls due to the access contention for the Flash memory is reported. The third column shows the clock cycles corresponding to stalls due to access contention

TABLE III
PERFORMANCE COUNTERS VALUES FOR THE TRIPLE-CORE SCENARIO

Decentralized Scheduler	Flash Memory Stalls [clock cycles]	System RAM Stalls [clock cycles]
DS1	1,878,336	663,386
DS2	1,932,409	791,922
DS3	1,589,729	478,264
DS4	1,738,412	525,011
DS5	1,929,209	788,668

for the system RAM. It is worth noting that any multi-core scheduler is limited by the shared memory architecture: as an example, considering exclusively the Flash memory, when moving from a single-core implementation to a multi-core one, the total number of clock cycles stalls increased from 200,679 to 1,589,729 (with the *DS3* scheduler). Hence, as confirmed by the values present in the second column of Table III, the Flash memory represents the real bottleneck. In the considered architecture, the flash is divided into different partitions and it does not allow multiple reads from the same partition (the STL code is the same for the three processors): therefore, only one processor at a time can read from the flash. The scheduler *DS3* may seem the most promising one, since it reduces substantially the execution time in both dual-core and triple-core scenarios. However, further experiments showed that it could hardly be used in an industrial context. Although it meets some of the requirements (minimum system resources usage), it suffers from a non-deterministic execution time when the number of tests composing the *ShareResource* set varies.

The results of the third set of experiments are shown in the charts of Figures 3 and 4, for the dual and triple-core scenarios, respectively. In both cases, it was increased progressively the number of self-test procedures composing the *ShareResource* set, that is the set of procedures requiring the use of a shared resource. This was done adding one self-test procedure at a time, which was not originally part of the *ShareResource* set. It is noteworthy that this does not mean altering the total number of programs composing the STL, which is still 104.

For both figures, the orange line represents the execution time when increasing the size of the *ShareResource* set, inserting in a descending order the self-test procedures starting with the longest test programs in terms of duration (their duration is in the range 10,000 to 23,000 clock cycles). The blue one instead represents the behavior of the scheduler when increasing the size of the set, inserting in an ascending order the self-test procedures starting with the shortest test programs (duration ranging from 500 to 1,000 clock cycles). The red line represents the threshold imposed by the serial execution. Since the purpose is to show the indeterminacy of these approaches, it was decided not to consider more than 25 self-test procedure composing *ShareResource*.

It can be noticed comparing the two charts that the execution time is not predictable and it presents non-negligible oscillations. By monitoring the STL execution on each processor core using an external debugger (the Lauterbach Trace32

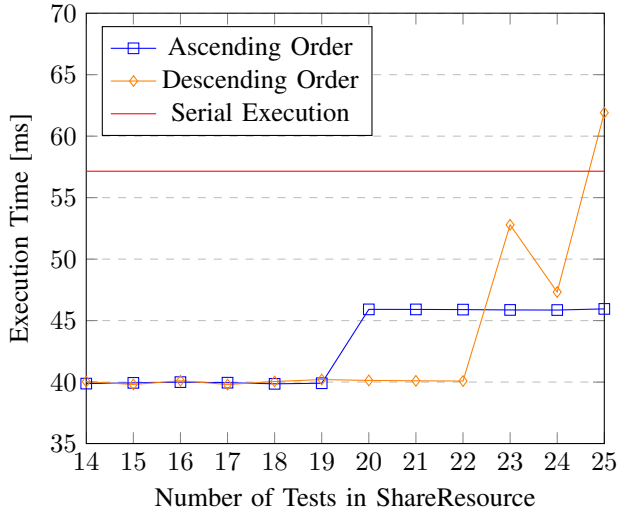

Fig. 3. *DS3* execution time (y-axis) at 16MHz when increasing the number (x-axis) of self-test procedures in the *ShareResource* set in a dual-core scenario.

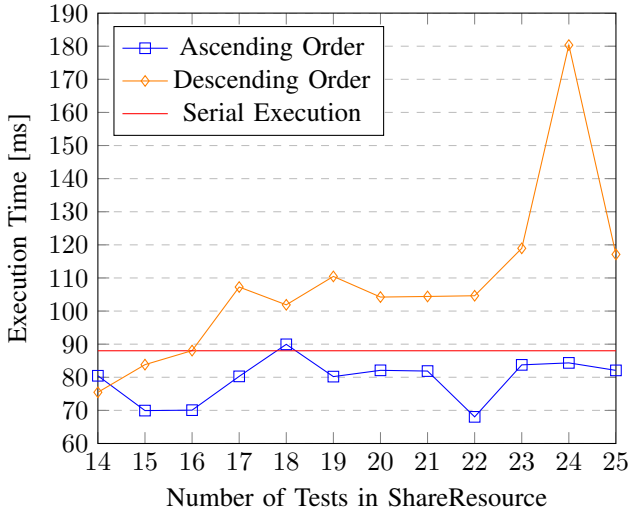

Fig. 4. *DS3* execution time (y-axis) at 16MHz when increasing the number (x-axis) of self-test procedures in the *ShareResource* set in a triple-core scenario.

in the performed experiments), it was observed a higher memory access contention. In particular, it emerged that these fluctuations are caused by the alternated execution of the test programs composing the *ShareResource* set with test programs not belonging to this set. Clearly, this is not acceptable: let us consider the scenario depicted in Figure 4, when moving from 19 to 18 self-test procedures in the *ShareResource* set. This may well be the typical scenario in which a customer, using the STL, decides to disable one test program. With 19 tests the execution time is below the threshold of the serial execution, but when reducing the tests the execution time actually increases above the serial execution.

D. The proposed selfish decentralized scheduler

The fourth set of experiments involves the proposed decentralized selfish scheduler described in the previous section. From the results presented in Table III, it appears that *DS3*

TABLE IV
PERFORMANCES OF THE PROPOSED DECENTRALIZED SELFISH SCHEDULER @16MHZ

# Active Cores	Execution Time [ms]		
	Serial Scheduler	DS3	Proposed Scheduler
2	57.15	39.87	38.27
3	88.01	66.80	57.18

TABLE V
OVERHEAD OF THE PROPOSED SCHEDULER

Overhead	Single-Core STL	Triple-Core STL
Memory Footprint [KB]	429	489
Execution Time [ms]	29.01	29.51

outperforms the other schedulers since it reduces the memory access contention. The reason for this reduction (confirmed again by monitoring the schedulers execution) originates from the fact that if the longest tests are executed first, it is likely that they will keep the shared resource busy for a considerable amount of time. This forces the other local schedulers to execute the self-test procedures not included in *ShareResource* and then wait for the shared resource to be freed. This significantly reduces the bus activity, but as the experiments of Figure 3 and 4 confirmed, it depends on the actual duration of the self-test procedures composing the *ShareResource* set.

The proposed scheduler enforces this condition with the devised order for *TestTable* and maintaining the resource busy until all the test programs in *ShareResource* are executed. The latter in particular avoids the alternated execution mentioned above, that causes the oscillations present in the scheduler *DS3*. Table IV reports the comparisons among the serial scheduler, the *DS3* scheduler and the proposed selfish scheduler when executing the STL with the original number of test programs in the *ShareResource* set (namely 13).

As it can be observed by comparing the second and the fourth column of Table IV, the proposed solution reduces considerably the execution time of about 33% and 35% (for the dual-core and triple-core scenarios respectively). This is significant, since especially in the triple-core scenario the execution time improves with respect to *DS3* of about the 14% and *it is comparable with a serial execution but in a dual-core scenario*. It is worth noting that the order of the self-test procedures within *ShareResource* is now irrelevant: the test programs within *ShareResource* are executed as an unique block.

Clearly, having multiple copies of the STL in the code memory would be beneficial for any multi-core scheduler. However, this is normally not possible when dealing with in-field test of embedded systems since this means a flash memory occupation two to three times higher than in a single-core scenario. As an example, considering the STL under analysis, the single-core version of the library occupies 429 Kbytes while the triple-core version is 489 Kbytes. Therefore, having independent copies of the single-core version is not acceptable since it leads to an excessive memory usage. On the other hand, the overhead from a timing point of view of the proposed scheduler is also modest, since it accounts for about $0.5ms$. Table V summarizes the main characteristics of

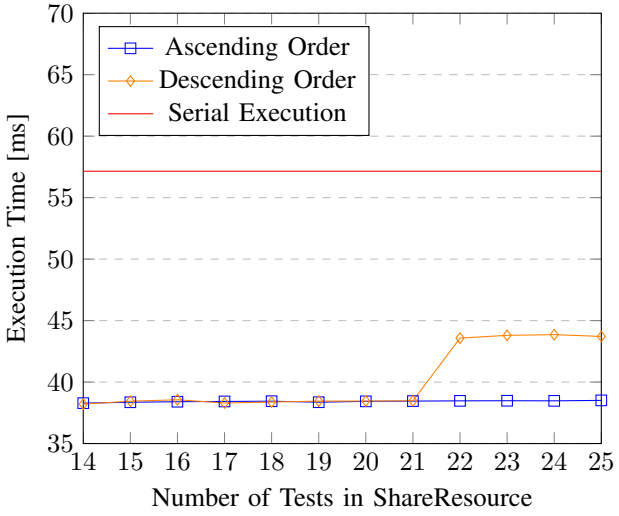

Fig. 5. The proposed scheduler execution time (y-axis) at 16MHz when increasing the number (x-axis) of self-test procedures in the *ShareResource* set in a dual-core scenario.

the proposed scheduler from a timing and memory footprint viewpoint for the triple-core scenario (being the worst possible in the considered case study). The reader should note that the execution time reported for the Triple-core STL is derived in a single-core scenario to show the overhead due to the decentralized scheduler structure, only.

The same experiments performed with *DS3* were repeated and the results are shown in Figure 5 and 6 for the dual and triple-core scenario, respectively. The behavior illustrated in Figure 5 and 6 is now much more predictable compared to the charts depicted in Figure 3 and 4. Furthermore, it can be seen that the execution time in the dual-core scenario (Figure 5) is always lower than the serial execution, unlike the behavior of *DS3* (Figure 3). It is important to note that in the triple-core scenario (Figure 6), the orange line after 21 self-test procedures in *ShareResource* crosses the red line. In more practical terms, it means that the execution time of the decentralized scheduler exceeded the serial scheduler. However, this represents an exaggerated case since the added test programs have a duration between 23,000 and 10,000 clock cycles. The reader should note that this is quite unrealistic in practical applications.

In order to asses the maximum achievable performances of the proposed scheduler, a further set of experiments focused on increasing the size of the *ShareResource* set, including progressively self-test procedures from *TestTable* not present originally in *ShareResource*. As in the experiments described in Figure 3, 4, 5 and 6, the self-test procedures were included starting from the shortest ones (in terms of duration) to the longest ones. However, the substantial difference with respect to the previous experiments is the fact that the aim is to increase the size of the *ShareResource* set as much as possible, well beyond the 25 self-test procedures of the aforementioned experiments. Figure 7 depicts the results of the experiments for two and three cores.

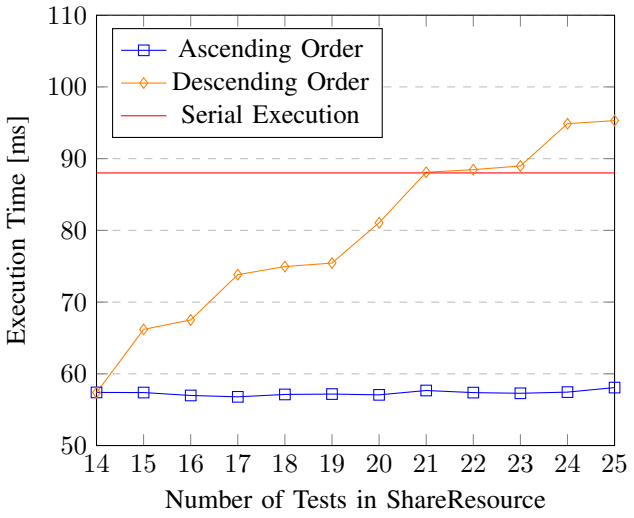

Fig. 6. The proposed scheduler execution time (y-axis) at 16MHz when increasing the number (x-axis) of self-test procedures in the *ShareResource* set in a triple-core scenario.

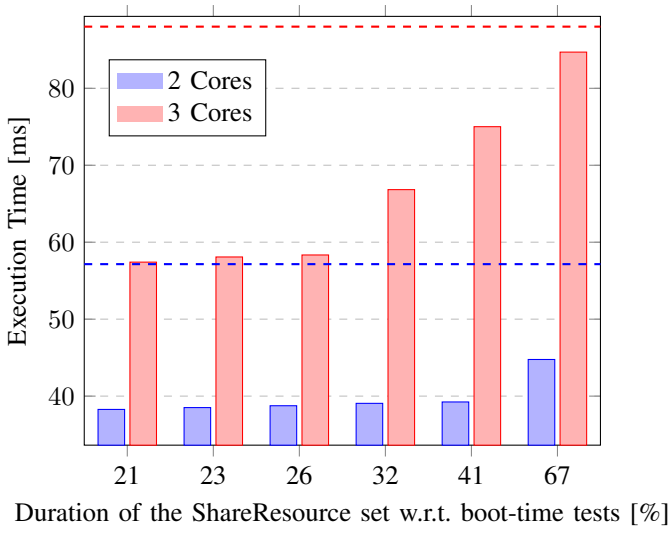

Fig. 7. The proposed scheduler execution time (y-axis) at 16MHz when increasing the duration (x-axis) of the *ShareResource* set. The blue and red dotted lines represent the serial scheduler for dual-core and triple-core scenarios, respectively.

For sake of generality, it is more convenient to express the results of Figure 7 using as x-axis the percent ratio between the duration of the *ShareResource* set and the total duration of the boot-time tests. It is important to underline that, in the considered STL, the tests labeled as boot-time are 70 out of 104. Therefore, 70 is also the maximum number of tests that can be included in the *ShareResource* set since the remaining 34 are run-time tests that are always present and by definition cannot be included in the *ShareResource* set. Figure 7 shows that up to a duration equal to 67% of the total duration of the boot-time tests, the execution time of the proposed decentralized scheduler is lower compared to a serial scheduler (for both dual-core and triple-core scenarios). It is noteworthy that a 67% figure corresponds to include in the *ShareResource* set 65 out of 70 self-test procedures. This means that it was possible to execute the vast majority of the test programs that can be labeled as boot-time tests. For completeness, the measurements corresponding to 90% and 100% of the boot-time duration were also gathered. In this case the execution time exceeded the one of the serial scheduler, since the last 5 test programs are the longest that can be included (each one requiring more than 15,000 clock cycles to execute). However, *it is uncommon having such long programs accessing the system RAM for testing purpose*. Typically, only few test programs require a shared portion of the system RAM for test purposes. Therefore, when considering a reasonable percent ratio of duration of the *ShareResource* set (namely 30-50%), the performances of the proposed decentralized scheduler are always superior compared to a serial scheduler.

By comparing the results shown in Figure 5, 6 and 7 it can be observed that as the size of the *ShareResource* set increases, the execution time of the decentralized scheduler degrades faster in a triple-core scenario than in the dual-core. This depends mainly from the fact that three active processors generate considerably more activity in the system bus than two processors. Moreover, as shown in Figure 2, in the considered architecture two processor cores (labeled as A and B) share the same cross bar (XBAR_2) while core C has a dedicated cross bar (XBAR_1). Therefore, it is inevitable that when all the three processors are active, the performances degrade faster since there are more conflicts on the same bus (not present when only two processors are active).

Finally, according to the development flow presented in [16], test programs are independently developed and then fault graded. Their cumulative effect is later considered for computing the final fault coverage figures. As it is extensively discussed in the above-mentioned paper, adopting the described development flow reduces the computational effort during the fault simulation process. Furthermore, another positive side effect stems from the fact that the computed fault coverage *does not depend on the actual test program order*. In particular, as long as the boot-time tests are not interrupted during their execution, the fault coverage is not altered. Therefore, since the proposed decentralized scheduler does not preempt the self-test procedures, the fault coverage is not altered.

VI. Conclusion

To the best of our knowledge, this paper describes for the first time a decentralized software scheduler for the concurrent execution of boot-time self-test procedures in a multi-core scenario targeting safety-critical applications. The proposed scheduler is based on a set of local schedulers, interacting each others by means of shared variables (i.e., a mutex). The benefits stemming from the adoption of the proposed scheduler are:

- *Maximum test concurrency*: local schedulers always try to execute other self-test procedures while the shared resource is busy.
- *Minimum system resource usage*: the proposed scheduler does not require having multiple copies of the STL

code in memory, nor different portions of the system RAM to be reserved for test purposes. Therefore, as the experiments confirmed, the overhead with respect to a single-core version of the STL is minimal.
- *Deterministic execution time*: when increasing (or decreasing) the number of test programs that cannot be executed concurrently, the execution time degrades (or improves) in a predictable manner (unlike other decentralized schedulers analyzed in the experimental section).
- *The fault coverage of the STL is not altered*: the test programs structure is not modified, nor the instructions stream (there is not preemption).
- *Full compliance* with the industrial STL development flow.

Since from the experimental results it emerged that the performances tend to degrade faster in a triple-core scenario, we are currently exploring different solutions for further improving the performances of the proposed decentralized scheduler in a scenario with three or more cores. Furthermore, we are also investigating the capability (and the related performances) of the proposed scheduler to handle different STLs, as in the case in which the SoC embeds processor cores of different types.

REFERENCES

[1] F. Reimann, M. Gla, J. Teich, A. Cook, L. R. Gmez, D. Ull, H. Wunderlich, U. Abelein, and P. Engelke, "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] E. Fujiwara, *Code Design for Dependable Systems: Theory and Practical Application*. New York, NY, USA: Wiley-Interscience, 2006.

[3] C. L. Chen and M. Y. Hsiao, "Error-correcting codes for semiconductor memory applications: A state-of-the-art review," *IBM Journal of Research and Development*, vol. 28, no. 2, pp. 124–134, March 1984.

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

[5] T. McLaurin, "Periodic online lbist considerations for a multicore processor," in *2018 IEEE International Test Conference in Asia (ITC-Asia)*, Aug 2018, pp. 37–42.

[6] M. Nicolaidis, "Theory of transparent bist for rams," *IEEE Transactions on Computers*, vol. 45, no. 10, pp. 1141–1156, Oct 1996.

[7] G. Harutyunyan, S. Shoukourian, V. Vardanian, and Y. Zorian, "An effective solution for building memory bist infrastructure based on fault periodicity," in *2013 IEEE 31st VLSI Test Symposium (VTS)*, April 2013, pp. 1–6.

[8] K. Darbinyan, G. Harutyunyan, S. Shoukourian, V. Vardanian, and Y. Zorian, "A robust solution for embedded memory test and repair," in *2011 Asian Test Symposium*, Nov 2011, pp. 461–462.

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

[10] A. Paschalis, D. Gizopoulos, N. Kranitis, M. Psarakis, and Y. Zorian, "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.

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

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

[13] N. Kranitis, A. Merentitis, G. Theodorou, A. Paschalis, and D. Gizopoulos, "Hybrid-sbst methodology for efficient testing of processor cores," *IEEE Design Test of Computers*, vol. 25, no. 1, pp. 64–75, Jan 2008.

[14] 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.

[15] L. Chen, S. Ravi, A. Raghunathan, and S. Dey, "A scalable software-based self-test methodology for programmable processors," in *Proceedings 2003. Design Automation Conference (IEEE Cat. No.03CH37451)*, June 2003, pp. 548–553.

[16] P. Bernardi, R. Cantoro, S. D. Luca, E. Sanchez, and A. Sansonetti, "Development flow for on-line core self-test of automotive microcontrollers," *IEEE Transactions on Computers*, vol. 65, no. 3, pp. 744–754, March 2016.

[17] P. Bernardi, R. Cantoro, S. De Luca, E. Sanchez, A. Sansonetti, and G. Squillero, "Software-based self-test techniques for dual-issue embedded processors," *IEEE Transactions on Emerging Topics in Computing*, pp. 1–1, 2018.

[18] (2019) Infineon Software Test Library: [Online]. Available: https://www.hitex.com/tools-components/software-components/selftest-libraries-safety-libs/pro-sil-safetcore-safetlib/.

[19] (2019) Cypress Software Test Library: [Online]. Available: http://www.cypress.com/file/249196/download.

[20] (2019) Renesas Software Test Library: [Online]. Available: https://www.renesas.com/en-eu/products/synergy/software/add-ons.html#read.

[21] (2019) Microchip Software Test Library: [Online]. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/52076a.pdf.

[22] (2019) ARM Software Test Library: [Online]. Available: https://www.arm.com/products/development-tools/embedded-and-software/software-test-libraries.

[23] 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.

[24] N. Bartzoudis, V. Tantsios, and K. McDonald-Maier, "Dynamic scheduling of test routines for efficient online self-testing of embedded microprocessors," in *2008 14th IEEE International On-Line Testing Symposium*, July 2008, pp. 185–187.

[25] D. Gizopoulos, "Online periodic self-test scheduling for real-time processor-based systems dependability enhancement," *IEEE Transactions on Dependable and Secure Computing*, vol. 6, no. 2, pp. 152–158, April 2009.

[26] M. A. Skitsas, C. A. Nicopoulos, and M. K. Michael, "Exploration of system availability during software-based self-testing in many-core systems under test latency constraints," in *2014 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, Oct 2014, pp. 33–39.

[27] M. Haghbayan, A. Rahmani, A. Miele, M. Fattah, J. Plosila, P. Liljeberg, and H. Tenhunen, "A power-aware approach for online test scheduling in many-core architectures," *IEEE Transactions on Computers*, vol. 65, no. 3, pp. 730–743, March 2016.

[28] 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.

[29] A. Apostolakis, M. Psarakis, D. Gizopoulos, A. Paschalis, and I. Parulkar, "Exploiting thread-level parallelism in functional self-testing of cmt processors," in *2009 14th IEEE European Test Symposium*, May 2009, pp. 33–38.

[30] N. Foutris, M. Psarakis, D. Gizopoulos, A. Apostolakis, X. Vera, and A. Gonzalez, "Mt-sbst: Self-test optimization in multithreaded multicore architectures," in *2010 IEEE International Test Conference*, Nov 2010, pp. 1–10.

[31] A. Floridia, D. Piumatti, E. Sanchez, S. De Luca, and A. Sansonetti, "Parallel software-based self-test suite for multi-core system-on-chip: Migration from single-core to multi-core automotive microcontrollers," in *2018 13th International Conference on Design Technology of Integrated Systems In Nanoscale Era (DTIS)*, April 2018, pp. 1–6.

[32] P. Bernardi, L. Ciganda, M. de Carvalho, M. Grosso, J. Lagos-Benites, E. Sanchez, M. Sonza Reorda, and O. Ballan, "On-line software-based self-test of the address calculation unit in risc processors," in *2012 17th IEEE European Test Symposium (ETS)*, May 2012, pp. 1–6.