Integrating Online Safety-related Memory Tests in Multicore Real-Time Systems

Integrating Online Safety-related Memory Tests in Multicore Real-Time Systems

Ciro Donnarumma†, Alessandro Biondi†, Francesco De Rosa, and Stefano Di Carlo‡
Rete Ferroviaria Italiana S.P.A., Italy
†Scuola Superiore Sant'Anna, Pisa, Italy
‡Politecnico di Torino, Torino, Italy
Email: {ciro,donnarumma, alessandro.biondi}@santannapisa.it

***Abstract*—Almost all functional safety standards that regulate safety-critical domains impose to periodically test hardware platforms at run-time. RAM memories are among the fundamental components of computing platforms and are notably subject to faults. Hence, they are also primary components to be tested. Unfortunately, RAM tests are destructive, require to be atomically executed, and are not cheap from a computational perspective. As such, if not properly managed, they can jeopardize the timing performance of a real-time system, especially when running upon a multicore platform.**

This paper proposes a software architecture to integrate online memory tests on multicore real-time systems. Furthermore, by jointly considering a task model and a safety model based on the EN50129 safety standard, it presents an approach to compute the optimal configuration of memory tests that preserves the system schedulability and guarantees a given tolerable functional failure rate (TFFR). Experimental results show that the proposed approach allows achieving a marginal impact on schedulability while preserving a TFFR that is compatible with the highest safety integrity level specified by the EN50129.

I. Introduction

Embedded computing systems with real-time requirements play a crucial role in a multitude of safety-critical systems [1]. They are employed in several domains such as railway, automotive, and avionics, where a failure of the control software can determine catastrophic events or produce severe damages to the surrounding environment or the system itself. For instance, a failure in the railway signaling system could result in a loss of human lives, whereas the failure of a satellite control system could destroy the satellite itself. Although in the latter case there is no loss of human lives, there is an unacceptable loss of a huge amount of money. To avoid these kinds of issues, domain-specific safety standards have been established and safety-critical embedded systems must comply with them to be deployed. These standards typically define a set of requirements and techniques that safety-critical embedded systems must implement to minimize the probability of catastrophic events.

One of the most common techniques suggested by the safety standards is the *redundancy* (also called composite fail-safe [2]). This technique consists in the replication of the whole processing system with the following objectives:

- *Fault detection*, i.e., the replication is useful to detect the failures of a replica to bring the system into a safe state (e.g., fail-stop if it is possible to switch off the system) when a fault is detected;
- *Fault tolerance*, i.e., if a failure occurs in one replica, another one can safely continue to offer the service (useful when it is not safe to switch-off the system).

To name a relevant case, the redundancy scheme that is typically used in the railway domain is the 2-out-of-2 (2oo2) composite fail-safe. This redundancy scheme is composed of two independent replicas executing the same operations over time. Non-restrictive activities can progress only if both replicas agree. Although this scheme allows detecting a failure in one replica, it could be not enough. A hazardous fault could be *dormant*, i.e., not detectable through the replication, for a long time, hence causing errors under particular conditions only. The longer the time a fault is dormant, the higher the probability that a co-incident fault also occurs in the other replica. In this way, the probability that the two replicas agree on an unsafe operation could become very high. Online periodic testing has to be performed to reduce the time needed to detect a fault to an acceptable level [2], hence reducing the probability that a fault could appear in both replicas.

The EN50129 [2] and EN60730 [3] safety standards, respectively for the railway domain and the domotics domain, describe the failure modes of the components belonging to an integrated circuit, along with techniques to detect such failures using periodic online testing. Memories are designed very tightly to the technology limits and are among the components that are most prone to faults [4]. The EN50129 and EN60730 standards distinguish between two types of memory areas:

- *Invariable memory*, that is a memory area of a processing system whose contents are not expected to change during the program execution (e.g., ROM or the text area in RAM).
- *Variable memory*, that is a memory area of a processing system whose contents are expected to change during the program execution (e.g., global data or stack areas in RAM).

Periodic online testing of invariable memories aims at detecting all faults affecting constant data in memory. This is typically done using check-sums or the replication and comparison of the invariable address space.

Differently, variable memory tests write a known data pattern in memory, perform some data manipulation, and finally check that the memory content is consistent [5]. This means that these tests overwrite the content of the memory (e.g., data used by a task), which must hence be restored

once the test completes. Consequently, they are said to be *destructive*.

Conversely, periodic online testing of variable memory aims at detecting more complex faults (specifically, those that pertain to the so-called DC fault model for data and address [4], [5]) such as *stuck-at* faults, in which a memory cell is locked to certain value, *stuck-open*, in which a cell cannot be accessed due to e.g. an open word line or an open bit line, open or high impedance outputs, as well as short circuits between signal lines.

To the best of our knowledge, almost all safety standards require the execution of the periodic online memory tests of this kind. To name two relevant examples, this is the case for the EN50129 in the railway domain and the EN60730 for the domotics domain. Furthermore, also the AUTOSAR standard for the automotive domain provides a detailed description of how these tests must be structured and executed by the Electronic Control Units (ECUs) present on a car [6]. To our records, online memory tests have been deployed in several industrial safety-critical systems, but almost all of them considered uniprocessor platforms and possibly cyclic task scheduling.

Unfortunately, integrating online memory tests in a multicore real-time system based on fixed-priority scheduling is not straightforward, both from a system-level and design perspective, and no previous work addressed this problem.

Contribution. This work tackles this issue by making the following contributions. First, it proposes a software architecture to integrate online memory tests on a fixed-priority, multicore real-time system. The architecture takes into account technological constraints of modern embedded multicore platforms and implementation issues are also discussed. Second, by jointly considering a real-time task model and a safety model based on the EN50129 standard, it presents an approach to compute the optimal configuration of a memory test that preserves the system schedulability and matches a safety requirement (in terms of tolerable functional failure rate). The approach is based on schedulability analysis, where the timing parameters of the memory test are linked to a safety requirement. Finally, experimental results to assess the performance of the proposed approach are presented, both in terms of schedulability and tolerable functional failure rate. The experiments are based on the profiling of the March-SS memory test [7] running on a Cortex-A53 processor.

Paper structure. The rest of this paper is organized as follows. Section II details the problem addressed in this work. Section III presents the model adopted in this work, which includes both a task and platform model, and a safety model based on the EN50129 standard. Section IV presents the proposed software architecture to integrate online memory tests in a multicore real-time system. Section V proposes an analysis-driven approach to configure the memory test. Section VI presents the experimental results. Section VII discusses the related work, and Section VIII concludes the paper.

II. Problem definition

This section describes a set of key properties of typical online memory tests proposed in previous work. Such properties are later used to delineate the challenges that have to be faced to integrate memory tests in a multicore real-time system. RAMs, i.e., variable memories, are considered.

A. How tests for RAMs work

March tests are the most popular algorithms used to perform the testing of RAMs. A march test consists of a well-ordered finite sequence of march-elements $\{M0; M1; \dots\}$. Each march element M_i denotes a finite ordered sequence of read and/or write operations to be performed on each memory location. A march element M_{i+1} can be executed only once all the operations of the previous march element M_i have been applied to each memory location. A march element can be executed in an increasing ($\Uparrow$), decreasing ($\Downarrow$), or irrelevant ($\Updownarrow$) order of memory addresses [8], [9].

A real example of march test for RAMs is MATS+ [10], which is typically described by the following sequence of three march-elements:

$$\text{MATS+} = \{\Updownarrow (w0); \Uparrow (r0, w1); \Downarrow (r1, w0)\},$$

where $w0$ and $w1$ denote write operations of values 0 and 1, respectively, and $r0$ and $r1$ denote read and verify operations of the expected expect values 0 and 1, respectively. The above notation indicates that MATS+ first writes 0 into all the memory cells with an irrelevant order ($\Updownarrow (w0)$). Subsequently, it reads every memory cell starting from the last one verifying that it contains 0, and then writes 1 ($\Uparrow (r0, w1)$). Finally, it reads each memory cell starting from the first one, verifying that it contains 1, and then writes 0 ($\Downarrow (r1, w0)$). From the above description, it is easy to understand that march tests always have a linear time complexity $O(n)$, with n being the number of cells in the memory. The linear time complexity is notably the most attractive feature of these algorithms [8].

B. Properties of RAM tests

RAM tests are characterized by three main properties. **Atomicity** is the most relevant one. As it is described in the previous section, a RAM march test applies a pattern of operations onto a sub-set or all memory locations. To ensure the validity of the result, a test for variable memories must be atomic in the sense that it needs to access the memory under test in an exclusive way. From a scheduling perspective, this implies that the test executes:

- in a non-preemptive fashion with respect to the other tasks running on the same core;
- in a non-preemptive fashion with respect to the interrupt service routines (ISRs) that are potentially executed on the same core; and
- without the interference of the other cores, as they can issue accesses to the memory that can jeopardize the test.

Furthermore, as these tests generally interest the whole RAM (or at least a consistent portion of it), they tend to require a large amount of time to execute, especially if considering modern embedded platforms that could likely dispose of a considerable amount of RAM memory (in the order of some GBs, as it happens in the railway domain). Hence, **computational heaviness** is the second property of RAM tests.

Finally, variable memory tests are destructive. Indeed, they write a known data pattern in memory, perform some data manipulation, and finally check that the memory content is consistent. This means that RAM tests overwrite the content of the memory (e.g., data used by a task), which must hence be restored once the test completes. Consequently, such tests are said to be characterized by a **destructiveness** property.

C. Issues arising from the integration of RAM tests

Due to the atomicity property, a lot of computational power is wasted during the test execution. Indeed, in a multicore platform, while one core is testing the RAM, all the other cores shall be halted. Furthermore, interrupts cannot be served, so their latency increase up to the test duration. These problems are further exacerbated because the tests are time-consuming (computational heaviness). Clearly, a plain implementation of such tests in which, once the test starts, the whole memory is tested is not acceptable for a real-time system, as a huge non-preemptive blocking would be generated to all tasks and ISRs in the system. The only way to overcome this issue is to partition the memory into several *segments* of equal size S_{SIZE} and testing them in an independent fashion. In doing so, the duration of atomic, non-preemptive executions is reduced to the time needed to test one memory segment only.

It is worth observing that variable memory tests typically detect also inter-word faults (i.e., write or read operations on a memory word causing a change of the content of other words) due to coupling issues among words [4]. Unfortunately, the partitioning of the memory into disjoint memory segments limits the detection of inter-word faults, as only those that affects words belonging to the same segment can be detected. Nevertheless, it is very likely that inter-word faults affect words that are mapped onto close addresses [4], [5]. Therefore, a memory partitioning scheme that splits the memory into several *overlapped* segments (still of size S_{SIZE}) significantly improves the inter-word faults coverage. The last of such overlapping segments covers both a portion at the end and the beginning of the memory (wrap around). This scheme is the one considered in this work and is illustrated in Figure 1.

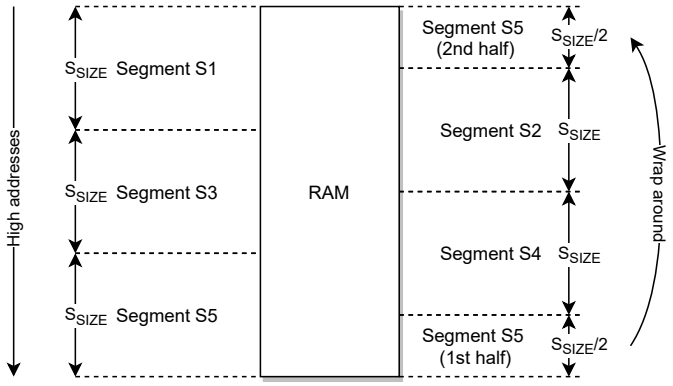

Fig. 1. Illustration of the memory partitioning with overlapped memory segments to be tested.

To address the destructiveness property, the content of a segment has to be saved before the test is executed on it, and eventually restored after the test completes its execution. In this way, the test becomes transparent to the application. Anyway, there is still another subtle problem related to destructiveness. Indeed, there will always be a critical set of memory segments that store the instructions and/or the data of the test routine itself. Clearly, the instruction and data fetched by the CPU to perform the test must not be corrupted by the test itself. This problem can be solved by *replicating* the test routine (instructions and data) into different memory segments. The memory is usually tested by the primary copy of the test, while the memory storing the primary is tested by the secondary copy. The two copies must be placed in non-adjacent segments in order to also test the memory segment that half overlaps a segment used by primary and half a segment used by secondary. An iteration of the test algorithm could otherwise destroy part of the content of both the copies.

D. Behavior of RAM tests

Given the partitioning of the memory into segments of size S_{SIZE} as discussed above, it holds that the larger the size of the segments:

- the higher the duration of the atomic (non-preemptable) test of a single segment;
- the higher the interrupt latency;
- the higher the inter-word fault coverage;
- the lower the overhead and the time needed to test the whole memory.

Hence, the memory segment size S_{SIZE} is a crucial parameter that influences several performance aspects of the system.

As we have seen in the last section, to address the destructiveness property, the test of each memory segment S_{UT} (segment under test) has to be arranged into three phases:

1) the content of S_{UT} is copied into a free memory segment S_{BK} (backup segment).
2) the test algorithm destroys the content of S_{UT}, verifying that it is fault free.
3) the content of S_{UT} is restored, copying it back from S_{BK}.

III. System model

A. Platform and real-time task model

This work considers a hardware platform consisting of a set $\mathcal{P} = \{P_1, \ldots, P_m\}$ of m homogeneous cores (or processors) sharing a memory of size M. The memory can be tested with a minimum granularity S_{STEP}, i.e., each memory segment used by the test must have size $S_{\text{SIZE}} = k \cdot S_{\text{STEP}}$, with $k \in N_{\geq 1}$. Each core P_k executes a set $\Gamma_k = \{\tau_1, \ldots, \tau_{n_k}\}$ of n_k periodic (or sporadic) tasks. Each task $\tau_{i,k}$ is characterized by a worst-case execution time (WCET) $C_{i,k}$, a release period $T_{i,k}$ (or minimum inter-arrival time), and a constrained relative deadline $D_{i,k} \leq T_{i,k}$. The utilization of task set Γ_k is denoted by $U_{\Gamma_k} = \sum_{\tau_{i,k} \in \Gamma_k} (C_{i,k}/T_{i,k})$. Tasks are scheduled according to partitioned fixed-priority scheduling, i.e., each task is always executed by the same core (it cannot migrate) and on each core the tasks are scheduled according to a fixed-priority algorithm.

Each task $\tau_{i,k}$ is assigned a priority $p_{i,k}$ and can be blocked by lower-priority tasks for at most $B_{i,k}$ time units (e.g., due

to a locking protocol, non-preemptable sections, etc.). For the sake of simplicity, we consider that for each P_k the tasks belonging to Γ_k are ordered with a decreasing priority order (i.e., $p_{i,k} \geq p_{i+1,k}$). Finally, we denote by $R_{i,k}$ an upper bound on the worst-case response time of $\tau_{i,k}$ and by $hp(\tau_{i,k})$ the set of higher-priority tasks with respect to $\tau_{i,k}$ running on P_k.

B. Safety model

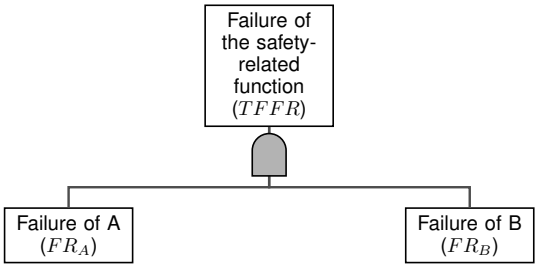

Fig. 2. Fault Tree of 2oo2 composite fail-safe architecture. A and B are two replicas executing the same safety-related function. The system fails in a hazardous way if both replicas experience a fault at the same time.

A safety-related function is a system function performing elaborations related to the safety of the environment in which the system operates. Any safety-related function has an associated Tolerable Functional Failure Rate (TFFR), which is the maximum rate at which the function itself can experience failures. The TFFR is evaluated by the risk analysis of the function. For instance, according to the EN50126 and EN50129 standards [2], [11] (adopted in the railway domain), a safety-related function with an integrity level SIL 4 must satisfy $10^{-9} \leq \text{TFFR} < 10^{-8}$.

In the railway domain, safety-related functions are typically performed by a 2oo2 composite fail-safe system. Such a kind of system is composed of two replicas: A, with a failure rate FR_A, and B, with a failure rate FR_B. Each of them is a sub-system compliant to the model described above and independently performing the same safety-related function (independence is necessary to avoid common-cause failure). The results of the safety-related function evaluated by replica A are compared against the results of the safety-related function evaluated by replica B. Only if there is an agreement on these results, the system can rely on them continuing its elaboration. If the results provided by the two replicas are different, an error has been detected, so the system goes into a fail-safe state. This safety mechanism allows the detection of errors, i.e., incorrect results produced by the safety-related function. Fig. 2 shows the fault-tree of this kind of system architecture in lack of common-cause failure.

A 2oo2 architecture can detect only the faults affecting the results produced by one of the two replicas. Anyway, a fault could be *dormant*, i.e., not affecting the result of the safety-related function. For instance, if the safety-related function relies on a flag stored in a memory cell affected by the stuck-at-0 fault (the cell is locked to 0) and the value of such a flag is actually 0, the fault will not affect the results evaluated from the function. This is the case of a dormant fault that is not detectable by the comparison made by the 2oo2 architecture. If a fault stays dormant for a long time, there exists the possibility that the same fault (e.g, stuck-at-0 on the same memory cell storing a flag of interest) occurs in the other replica. In this situation, the 2oo2 architecture could rely on a wrong result of the safety-related function leading to unsafe operations. This means that any dormant fault shall be detected, and a safe-state enforced, in a time that is sufficiently short to ensure that the risk of a second fault occurring during the detection time is smaller than the specified probabilistic target given by the TFFR.

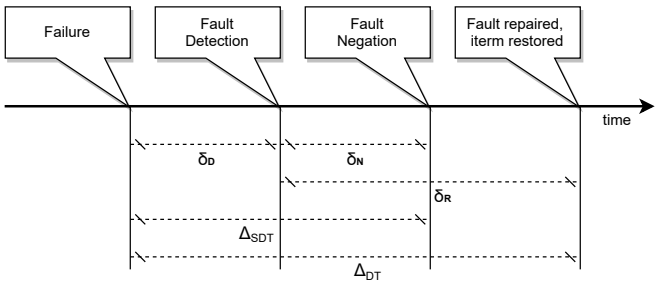

Fig. 3. Life-cycle of a fault according to the EN50129 standard.

Fig. 3 shows the typical life-cycle of a fault according to the EN50129 standard [2]. As soon as a failure comes out, it is detected in a detection-time δ_D. Then, some countermeasures are taken to negate the fault, bringing the system into a safe state.

The time δ_N required to negate a fault is called *negation time*, while the time δ_R between the detection of the fault and its repair is called *repair time*. Once the fault is negated, the system enters into a state of safe operation. Instead, once the fault is repaired, the system will enter again into its state of normal operation. The total time $\Delta_{\text{SDT}} = \delta_D + \delta_N$ between the occurrence of a failure and its fault negation is called *safe down-time*.

In this work, we consider the same safety model specified by the EN50129 [2] standard. Specifically, it is assumed that the system has constant failure rates (i.e., the number of failures per unit of time) FR over time, and that all faults are detected and have to be negated (with the exception of hazardous common cause failures, which must not be present by construction [2]). Still following the EN50129, we assume that the failure rate is much smaller than the safe down rate ($1/\Delta_{SDT}$), i.e., $FR \times \Delta_{\text{SDT}} << 1$, and that the detection time is larger than the negation time, i.e., $\delta_D >> \delta_N$, which implies that the safe down-time can be approximated with the detection time, i.e., $\Delta_{\text{SDT}} \approx \delta_D$.

A fault can be detected at the earliest when the test starts and at the latest when the test completes. As considered by the EN50129, given the time Δ_T that a test needs to check the whole memory, a fault is detected on average in $\Delta_T/2$ time units, i.e., $\delta_D = \Delta_T/2$. Under the previous assumptions, the EN50129 standard defines an upper bound for the time Δ_T within which the whole memory has to be tested, that is

$$\Delta_T < \Delta_T^{\max} = \frac{\text{TFFR}}{(FR_A \times FR_B)} \tag{1}$$

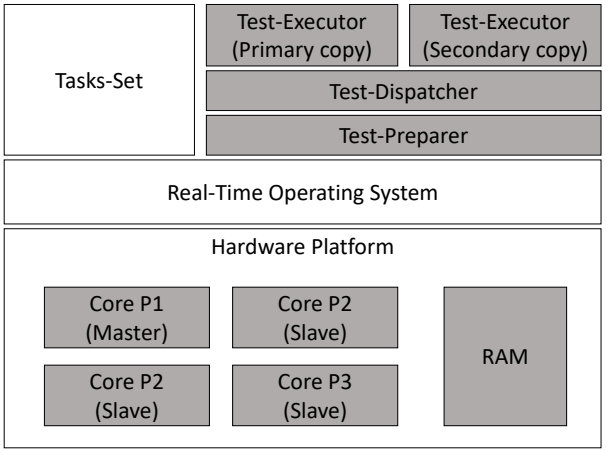

Fig. 4. Illustration of the proposed software architecture for memory testing on a 4-core platform.

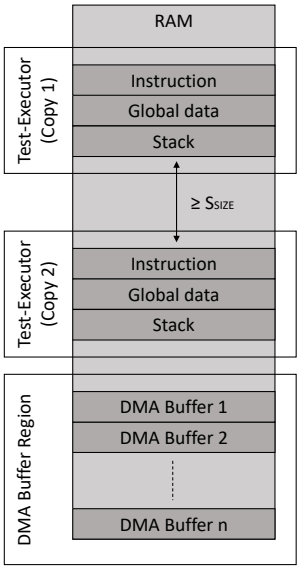

Fig. 5. Memory layout of the proposed software architecture.

IV. Proposed software architecture

This section presents a software architecture to integrate RAM tests in multicore real-time systems scheduled by fixed-priority partitioned scheduling. The architecture is illustrated in Fig. 4. To ensure the atomicity property of memory tests, the architecture splits the processor cores into two types:

- the *master core* (P_1, without loss of generality), which is in charge of executing the memory test algorithm; and
- the *slave* cores (i.e., all the other cores of the platform) that, when the test is activated, busy-wait for the completion of the test algorithm on the master core to avoid interfering with the execution of the test.

The proposed software architecture also splits the logic behind the memory tests into two components:

- The *Test-Preparer*, which selects the next memory segment to test, evaluating its base address and its size. This functionality is replicated over all the cores of the platform.
- The *Test-Executor*, which executes the test algorithm on the memory segment selected by the Test-Preparer. It is executed only on the master core.

The Tests-Preparer component knows the memory layout of the system. In some platforms, multiple RAM banks or different portions of the same memory bank are mapped onto the addressing space seen by the processor in a non-contiguous fashion. Therefore, the Tests-Preparer must dispose of an internal description of the memory layout, which is called RAM-descriptor. The RAM-descriptor is a list of all the memory blocks that compose the whole RAM addressing space, as seen by the processor. A memory block is a portion of the RAM that is contiguously mapped onto the processor addressing space. Hence, the RAM-descriptor contains the start address and the size of all the memory blocks.

For each core, the real-time operating system (ROTS) is in charge of periodically activating a test task every T_S time units, which runs at the highest priority in the system with disabled interrupts. Each job of a test task serves the purpose of testing *one* memory segment. The test tasks are synchronously released on all cores.

Each test task first executes the Test-Preparer, which selects a memory segment with size S_{SIZE} from a memory block to be tested, hence issuing a *test request* defined by the base address and the size of the selected segment. When all the segments composing a block have been tested, the Test-Preparer selects the next block in the RAM-descriptor. When all the segments of all the blocks are tested, the Test-Preparer restarts the selection from the first block.

After completing the execution of the Test-Preparer, the test task running on the master core executes the the Test-Executor, which is a component that performs these actions in the following order:

1) Inter-core synchronization: it busy-waits until *all* the test tasks running on the slave cores completed the execution of the Test-Preparer (to ensure atomicity).
2) Backup of the segment under test to avoid data loss (remember that the test is destructive).
3) Execution of the test algorithm on the segment under test selected by the Test-Prepararer.
4) Restoration of the content tested segment.
5) Wake-up of the slave cores: it sends a notification to the slave cores to signal the completion of the memory test.

Conversely, still after completing the execution of the Test-Preparer, the test tasks on the slave cores **(i)** first send a notification to the master core (for the purpose of the inter-core synchronization mentioned above); and then **(ii)** busy-wait

until the completion of the one on the master core, i.e., after the completion of the Test-Executor. An example schedule of the test tasks on a dual-core system is illustrated in Figure 6.

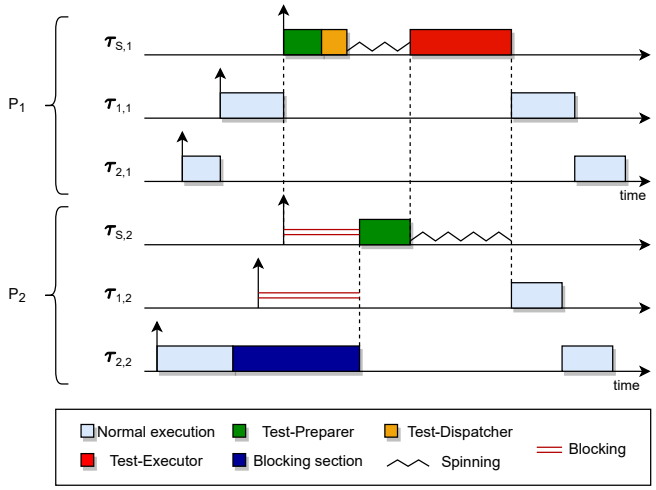

Fig. 6. Example schedule on dual-core system with the test tasks.

The busy-waiting employed by the Test-Executor can be implemented via classical shared-memory spin locks, as it occurs before starting the memory test. Differently, to avoid issuing memory accesses that interfere with the test, the busy-waiting performed by the slave cores has to be implemented by spinning on a processor register and waiting for an inter-core interrupt to exit the spinning. In this case, the fetching of the spinning instruction is the only thing that may interfere with the test (note that the ISR of the inter-core interrupt is executed when the test is completed): its interference can be mitigated by waiting the maximum time to cache the instruction before starting the Test-Executor or via a cache lock-down mechanism.

As mentioned in the previous sections, the whole RAM is treated as variable memory. This is true also for the portion of memory storing the code and the instruction of the Test-Executor itself. In order to also test these segments, the Test-Executor has to be *duplicated* in memory (both for the master core and the slave cores). Each copy (replica) of the Test-Executor disposes of its own private instruction area, global data area, and stack, so that all the memory accesses are restricted to its private memory only. The resulting memory layout is illustrated in Figure 5.

A dispatching component to select the copy of the Test-Executor to be used is hence needed. The Test-Dispatcher lives between the Test-Preparer and the two copies of the Test-Executor. When the Test-Preparer issues a test request, the Test-Dispatcher performs the following operations: if the test request does not interest one of the memory segments used by the primary copy of the Test-Executor, then it forwards the request to the primary copy; otherwise, it forwards the request to the secondary copy.

Note that the proposed software architecture is also applicable to systems that dispose of peripheral devices with direct memory access (DMA). These devices are programmed by the cores to autonomously access a buffer stored in RAM. For example, this is the case for Ethernet devices that, due to their high communication throughput, have a DMA engine that stores the frames received by the network interface into a memory buffer, without any intervention of the cores. To avoid data loss (e.g., incoming Ethernet frame), such devices cannot be halted, and hence memory tests cannot be performed on the memory regions used by such devices. Nevertheless, safety-related communications must always be protected by error-checking mechanisms and error-correcting codes, therefore memory faults that affect such regions would be anyway detected by other software components. In this case, the proposed architecture can be used by confining the memory buffers accessed by DMA engines of peripheral devices in a memory region and removing that region from the RAM-descriptor of the Test-Preparer. Clearly, this will make impossible to detect the inter-word coupling faults between the memory cells belonging to the DMA buffers and the other ones. However, this is unavoidable without stopping the DMA engines, i.e., without accepting to lose communication data.

V. Configuring Memory Tests

This section presents a methodology to configure the memory test such that **(i)** all tasks in the system, i.e., all task sets Γ_k, are *schedulable*; and **(ii)** the safety bound given by Equation (1) is respected.

Following the architecture presented in the previous section, the memory test is implemented with a set of synchronously-released, periodic tasks, one for each core, running at the highest priority to ensure the atomicity property. The test task on core P_k is denoted by $\tau_{S,k}$ and its WCET is denoted by $C_{S,k}$. All test tasks are released with period T_S.

According to partitioned fixed-priority scheduling, a system can be deemed schedulable if all task sets of all cores are schedulable. Hence, by applying standard response-time analysis [12] to the application tasks on each core, a system can be deemed schedulable if

$$\forall P_k \in \mathcal{P}, \forall \tau_{i,k} \in \Gamma_k, R_{i,k} \leq D_{i,k}, \tag{2}$$

where $R_{i,k}$ is least positive fixed point of the recurrence

$$R_{i,k} = B_{i,k} + C_{i,k} + \sum_{\tau_{j,k} \in hp(\tau_{i,k})} \left\lceil \frac{R_{i,k}}{T_{j,k}} \right\rceil C_{j,k} + \left\lceil \frac{R_{i,k}}{T_S} \right\rceil C_{S,k}. \tag{3}$$

Schedulability of the test tasks should also be checked (i.e., they must complete before their next activation): this aspect is addressed in a following subsection.

The challenge faced in this work consists in configuring the parameters T_S and $C_{S,k}$, for $k = 1, \ldots, m$, such that both Equation (2) and the safety bound of Equation (1) are satisfied. We proceed by studying the dependency of parameters T_S and $C_{S,k}$ on the safety bound of Equation (1).

A. Impact of the safety bound

According to the safety model presented in Section III-B (that follows from the EN50129 standard), the test of the whole RAM must be completed every Δ_T time units, bounded by Equation (1). The memory partitioning scheme presented in

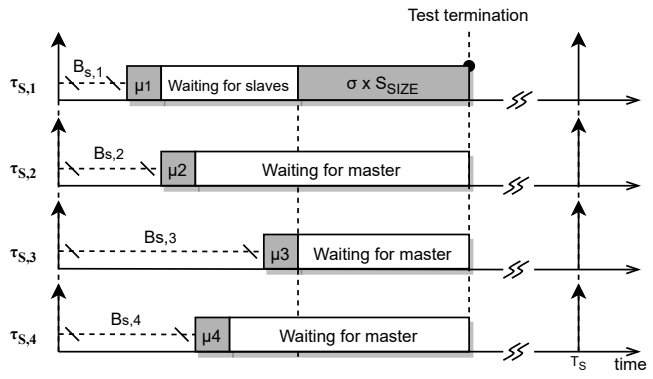

Fig. 7. Example schedule of the test tasks on a 4-core platform.

Section II-C splits the memory into N_S overlapped memory segments of size S_{SIZE}, where

$$N_S = \frac{2 \cdot M}{S_{\text{SIZE}}},$$

which allows decomposing the test of the whole memory into N_S shorter tests executed upon different memory segments. Each periodic instance of the test (specifically, each job of the test task running on the master core) performs one of such N_S tests. Given that the whole memory has to be tested in Δ_T time units, the period T_S of the test tasks must be

$$T_S = \frac{\Delta_T}{N_S} = \frac{\Delta_T \cdot S_{\text{SIZE}}}{2 \cdot M}. \quad (4)$$

Note that the above equation allows bounding the period of the test tasks with the safety bound of Equation (1), as it follows that

$$\frac{2 \cdot M \cdot T_S}{S_{\text{SIZE}}} < \Delta_T^{\max} = \frac{\text{TFFR}}{(FR_A \times FR_B)},$$

which implies

$$T_S < \frac{\Delta_T^{\max} \cdot S_{\text{SIZE}}}{2 \cdot M}. \quad (5)$$

B. Modeling the test tasks

Likewise the application tasks in the sets Γ_k, also the test tasks may suffer blocking, e.g., in the case in which one of the application tasks needs to execute a non-preemptive section. Hence, to adopt a more general model for the test tasks, we consider that each test task $\tau_{S,k}$ can be blocked by low-priority tasks by at most $B_{i,k}$ time units. An example schedule of the test tasks on a four-core platform is illustrated in Figure 7.

We proceed by studying the WCET of the test tasks. The test task $\tau_{S,1}$ running on the master core executes the components of the software architecture proposed in Section IV, i.e., it starts executing the Test-Prepararer, then the Test-Dispatcher, and finally the Test-Executor. Note that the execution of such a task can be split into three phases:

- *Test preparation*: the task executes the Test-Preparar and the Test-Dispatcher, hence executing a set of instructions in preparation of the actual test that are *independent* of the amount of memory S_{SIZE} to be tested;
- *Waiting for slave cores*: the task synchronizes with the other test tasks running on the slave cores by busy-waiting as discussed in Section IV;
- *Actual test*: the task performs the actual test of a memory region, hence executing for a time that is *proportional* to S_{SIZE}.

Conversely, the execution of the slave tasks can be split into two phases, i.e., the test preparation and the waiting for the master core.

Let μ_k denote the maximum duration of the test preparation phase for each task $\tau_{S,k}$. Also, let $\sigma \cdot S_{\text{SIZE}}$ be the maximum time required to test a memory region of size S_{SIZE}, where σ is a proportional factor that determines the "speed" of the test. The following lemma allows computing the WCET of each test task.

Lemma 1: The WCET of test task $\tau_{S,k}$ is given by

$$C_{S,k} = \max\left\{\mu_k, \max_{P_x \in \mathcal{P} \setminus \{P_k\}} \{B_{S,x} + \mu_x\}\right\} + \sigma \cdot S_{\text{SIZE}}. \quad (6)$$

Proof: Let us first consider the test task $\tau_{S,1}$ of the master core. Two execution scenarios for this task are possible: (i) the task does not wait for the completion of any another test task running on a slave core, and (ii) otherwise. In the first case, the execution of the task just comprises the test preparation and the actual test. Hence, its WCET is given by $\mu_k + \sigma \cdot S_{\text{SIZE}}$. In the second case, being the test tasks synchronously released, $\tau_{S,1}$ cannot wait more than the time that spans from its release to the latest completion of the test preparation phases of the other test tasks. Each slave task $\tau_{S,k}$, with $k > 1$, completes its preparation phase at the latest after having been blocked by the maximum time $B_{S,k}$ and having executed by at most μ_k time units. This means that, independently of the time it is blocked and the time it takes to execute its test preparation, task $\tau_{S,1}$ is ready to perform the actual test after at most $c_1 = \max_{P_x \in \mathcal{P} \setminus \{P_1\}} \{B_{S,x} + \mu_x\}$ time units following its release at time r. For the whole time interval $[r, r+c_1]$, $\tau_{S,1}$ can either execute the test preparation or busy-wait (note that it is always possible that the task is not blocked). Hence Equation (6) holds.

Now, consider the test slave tasks $\tau_{S,k}$ of the slave cores ($k > 1$). After executing the test preparation, such tasks have to wait for the completion of the test task on the master core. The actual test performed by the master core can start when all the test tasks complete their test preparation phases. All other test tasks $\tau_{S,x}$ with $k \neq x$, complete their test preparation phases no later than $c_k = \max_{P_x \in \mathcal{P} \setminus \{P_k\}} \{B_{S,x} + \mu_x\}$ time units after the release of $\tau_{S,k}$ at time r. If $\tau_{S,k}$ is blocked by $b \leq B_{S,k}$ time units and executes for $\nu \leq \mu_k$ time units, two scenarios are possible: (i) $c_k \geq b + \nu$, and (ii) otherwise. In case (i), as discussed for the test task on the master core, we have that for the whole time interval $[r, r + c_k]$ task $\tau_{S,k}$ can either execute or busy-wait. After time $r + c_k$, the actual test can start on the master core, hence determining that $\tau_{S,k}$ waits for other $\sigma \cdot S_{\text{SIZE}}$ time units. Note that the sum of these two contributions is matched by Equation (6). In case (ii), the actual test starts when $\tau_{S,k}$ itself completes its test preparation phase at time $r + b + \nu$. By definition, in time interval $[r, r + b + \nu]$ task $\tau_{S,k}$ executes for $\nu \leq \mu_k$ time units. Hence, the total execution

time of $\tau_{S,k}$ is bounded by μ_k plus the time to perform the actual test ($\sigma \cdot S_{\text{SIZE}}$). Hence the lemma follows. ■

Similarly, the schedulability of the test tasks can be checked as follows.

Lemma 2: The test tasks are schedulable if

$$\max_{P_x \in \mathcal{P}} \{B_{S,x} + \mu_x\} + \sigma \cdot S_{\text{SIZE}} \leq T_S. \tag{7}$$

Proof: Each test task $\tau_{S,x}$ can be blocked by at most $B_{S,x}$ time units and then can execute by at most μ_x time units before completing its test preparation phase. Hence, after $\max_{P_x \in \mathcal{P}}\{B_{S,x} + \mu_x\}$ time units from their release, all test tasks complete their test preparation phases. Subsequently, the actual test can take at most $\sigma \cdot S_{\text{SIZE}}$, after which all test tasks finish executing. Hence the lemma follows. ■

Thanks to Lemma 1 and Equation (4), it is possible to compute the maximum utilization of each test task $\tau_{S,k}$ as:

$$U_{S,k} = \frac{C_{S,k}}{T_S} = \frac{2 \cdot M}{\Delta_T} \cdot \left(\frac{L_{S,k}}{S_{\text{SIZE}}} + \sigma \right), \tag{8}$$

where

$$L_{S,k} = \max \left\{ \mu_k, \max_{P_x \in \mathcal{P} \setminus \{P_k\}} \{B_{S,x} + \mu_x\} \right\}. \tag{9}$$

C. Finding lower and upper bounds for S_{SIZE}

The above results can be used to constrain the design space that has to be explored to configure the memory test.

Lemma 3: The considered system can be schedulable only if the memory test is configured with $S_{\text{SIZE}} \geq S_{\text{min}}$, where

$$S_{\text{min}} = \left\lceil \frac{\max_{P_k \in P} \left\{ \dfrac{2 \cdot M \cdot L_{S,k}}{\Delta_T \cdot (1 - U_{\Gamma_k}) - 2 \cdot M \cdot \sigma} \right\}}{S_{\text{STEP}}} \right\rceil \cdot S_{\text{STEP}}. \tag{10}$$

Proof: A necessary condition for the system schedulability is that each core is not overloaded (i.e., utilization ≤ 1). Hence, the system can be schedulable only if, for each core P_k, it holds $U_{\Gamma_k} + U_{S,k} \leq 1$. By injecting Equation (8) in the latter inequality we get

$$U_{\Gamma_k} + \frac{2 \cdot M \cdot L_{s,k}}{\Delta_T \cdot S_{\text{SIZE}}} + \frac{2 \cdot M \cdot \sigma}{\Delta_T} \leq 1,$$

which can be re-arranged as

$$\frac{2 \cdot M \cdot L_{s,k}}{\Delta_T \cdot S_{\text{SIZE}}} + \leq 1 - U_{\Gamma_k} - \frac{2 \cdot M \cdot \sigma}{\Delta_T}$$

$$\Rightarrow \quad \frac{2 \cdot M \cdot L_{s,k}}{\Delta_T \cdot S_{\text{SIZE}}} + \leq \frac{\Delta_T \cdot (1 - U_{\Gamma_k}) - 2 \cdot M \cdot \sigma}{\Delta_T}$$

$$\Rightarrow \quad \frac{1}{S_{\text{SIZE}}} \leq \frac{\Delta_T \cdot (1 - U_{\Gamma_k}) - 2 \cdot M \cdot \sigma}{2 \cdot M \cdot L_{s,k}}.$$

Hence, for each core P_k, we have

$$S_{\text{SIZE}} \geq \frac{2 \cdot M \cdot L_{S,k}}{\Delta_T \cdot (1 - U_{\Gamma_k}) - 2 \cdot M \cdot \sigma}.$$

As the latter inequality must hold for each core, we have

$$S_{\text{SIZE}} \geq \max_{P_k \in P} \left\{ \frac{2 \cdot M \cdot L_{S,k}}{\Delta_T \cdot (1 - U_{\Gamma_k}) - 2 \cdot M \cdot \sigma} \right\}.$$

The lemma follows by noting that S_{SIZE} cannot be arbitrarily chosen, as it has to be a multiple of the granularity S_{STEP}. ■

Let $R'_{i,k}$ be an upper-bound on the worst-case response time of $\tau_{i,k}$ when considering task set Γ_k only, i.e., without the test task $\tau_{S,k}$ (it can be obtained from Equation (3) by simply omitting the last term in the sum).

Lemma 4: The considered system can be schedulable only if the memory test is configured with $S_{\text{SIZE}} \leq S_{\text{max}}$, where

$$S_{\text{max}} = \left\lfloor \frac{\min_{P_k \in P} \left\{ \dfrac{\min_{\tau_{i,k} \in \Gamma_k} \left\{ D_{i,k} - R'_{i,k} \right\} - L_{S,k}}{\sigma} \right\}}{S_{\text{STEP}}} \right\rfloor \cdot S_{\text{STEP}}. \tag{11}$$

Proof: Being the test tasks running at the highest priority on each core, in the worst case the application tasks of the sets Γ_k can be preempted at least once from a test task. Hence, it follows that the WCET of each test task $\tau_{S,k}$ must not be larger than the *slack* time of each task in Γ_k when the test task is not present. Otherwise, the introduction of the test task on core P_k will certainly cause a deadline miss.

Hence, for each core P_k, we have that a schedulable system must satisfy

$$\forall \tau_{i,k} \in \Gamma_k, C_{S,k} \leq D_{i,k} - R'_{i,k}.$$

This is equivalent to

$$C_{S,k} \leq \min_{\tau_{i,k} \in \Gamma_k} \{D_{i,k} - R'_{i,k}\}.$$

By Lemma 1 and Equation (9), the latter inequality becomes

$$L_{S,k} + \sigma \cdot S_{\text{SIZE}} \leq \min_{\tau_{i,k} \in \Gamma_k} \{D_{i,k} - R'_{i,k}\},$$

which can be rewritten as

$$S_{\text{SIZE}} \leq \frac{\min_{\tau_{i,k} \in \Gamma_k} \{D_{i,k} - R'_{i,k}\} - L_{S,k}}{\sigma}.$$

Therefore, S_{SIZE} cannot be larger than

$$S_{\text{max}} = \min_{P_k \in P} \left\{ \frac{\min_{\tau_{i,k} \in \Gamma_k} \{D_{i,k} - R'_{i,k}\} - L_{S,k}}{\sigma} \right\}. \tag{12}$$

The lemma follows by noting that S_{SIZE} cannot be arbitrarily chosen, as it has to be a multiple of the granularity S_{STEP}. ■

D. Computing the optimal configuration

Thanks to the above results, it is finally possible to compute the optimal configuration of the memory test via a design space exploration of the domain of parameter S_{SIZE}. Note that, as mentioned in Section II-D, the larger S_{SIZE} the better as more inter-word faults can be covered. Furthermore, the larger S_{SIZE} the lower the test is fragmented, hence reducing the overall overhead it introduces. Hence, the optimal S_{SIZE} is the maximum one that makes the system schedulable while preserving the safety bound of Equation (1).

Note that the lower bound provided by Lemma 3 is monotone decreasing with Δ_T. Hence, setting $\Delta_T = \Delta_T^{\max} - \epsilon$, with $\epsilon > 0$ arbitrarily small, in the lemma equation ensures both the safety bound of Equation (1) and a valid lower bound $S_{\min}$ for S_{SIZE} independently of the time taken to test the whole memory. Furthermore, being the period of the test tasks monotone increasing with Δ_T (see Equation (4)), setting $\Delta_T = \Delta_T^{\max} - \epsilon$ also yields the best schedulability performance for a given S_{SIZE} without violating the safety bound. Indeed, the safety bound is respected as long as $\Delta_T < \Delta_T^{\max}$ and choosing a lower Δ_T can only worsen the system schedulability due to sustainability of fixed-priority scheduling with respect to the tasks' periods.

Therefore, to compute the optimal configuration of the memory test, it suffices to explore all values of S_{SIZE} from $S_{\max}$ to $S_{\min}$ with step S_{STEP} and check whether the corresponding configuration is schedulable. This strategy is provided in Algorithm 1. Note that each value of S_{STEP} determines the WCET (by Lemma 1) and the period (by Equation (4)) of the test tasks, hence enabling the schedulability analysis of each core. The algorithm terminates as soon as a schedulable configuration is found. The sub-function IS_SCHED returns True if a task set is schedulable given the test of Equations (2)-(3), False otherwise.

Algorithm 1 Algorithm to compute the optimal configuration of the memory test while preserving schedulability and the safety bound.

procedure CONFIGMEMTEST($\cup_{P_k \in \mathcal{P}}\{\Gamma_k\}, M, S_{\text{STEP}}, \epsilon$)
  $\Delta_T = \Delta_T^{\max} - \epsilon$
  Compute $S_{\max}$ and $S_{\min}$ by Lemmas 3 and 4
  for S_{SIZE} **from** $S_{\max}$ **to** $S_{\min}$ **with step** $-S_{\text{STEP}}$ **do**
    Found $\leftarrow$ **True**
    $T_S \leftarrow (\Delta_T \cdot S_{\text{SIZE}})/(2 \cdot M)$
    for k **from** 1 **to** m **do**
      $L_{S,k} \leftarrow \max\{\mu_k, \max_{P_j \in P}(B_j + \mu_j)\}$
      $C_{S,k} \leftarrow L_{S,k} + \sigma \cdot S$
      $\tau_{S,k} \leftarrow$ **new** Task(C_S, T_S, T_S)
      Found $\leftarrow$ Found **AND** IS_SCHED(Γ_k, $\tau_{S,k}$)
    end for
    if $\max_{P_x \in \mathcal{P}}\{B_{S,x} + \mu_x\} + \sigma \cdot S_{\text{SIZE}} > T_S$ **then**
      Found $\leftarrow$ False
    end if
    if Found **then**
      return $\{S_{\text{SIZE}}, \cup_{P_k \in \mathcal{P}}\{\tau_{S,k}\}\}$
    end if
  end for
  return no solution found
end procedure

VI. EXPERIMENTAL RESULTS

The performance of the proposed approach has been evaluated through an experimental study based on synthetic workload. The goal of this study was to both assess the degradation of the schedulability performance of a system when integrating the memory test, and to quantify the Tolerable Functional Failure Rate (TFFR) that can be achieved while not jeopardizing the schedulability.

To this end, four experiments have been performed to measure the performance of the proposed approach with respect to **(i)** the system utilization, **(ii)** the number of cores, **(iii)** the parameter σ (that controls the "speed" of the test), and **(iv)** the TFFR.

Unless otherwise noted in the next sections, we considered the following system configuration. We considered a two-replica system where each replica disposed of a RAM with size $M = 2GB$. The failure rate of the RAM for both the replicated systems is $FR = FR_A = FR_B \approx 1.389 \cdot 10^{-14} Hz$, which implies $\Delta_T^{\max} \approx 10$ hours for TFFR $= 10^{-9}$. The periods of the tasks have been randomly generated in the range $[50\ ms, 1000\ ms]$ with log-uniform distribution, with a period granularity of $1ms$. Tasks have implicit deadlines. Furthermore, the number of tasks running on each core was randomly chosen from the range $[5, 10]$ with uniform distribution. Each task was assigned a maximum duration of a non-preemptive section randomly selected in the range $[0\ \mu s, 10\ \mu s]$ with uniform distribution (such sections determine the blocking times considered in the analysis). To guarantee the generation of plausible non-preemptive sections, if the generated duration was higher or equal than the task's WCET, it was considered $0\mu s$. We assigned to σ the fixed value $1.5\mu s$/bytes: this value was obtained by experimentally measuring the time taken by the March-SS test [7] executed on a Cortex-A53 core of the Zynq Ultrascale+ platform by Xilinx. Also, for each core P_k, we randomly selected $\mu_k \in [10\ \mu s, 200\ \mu s]$ with uniform distribution. Finally, we set $S_{\text{STEP}} = 512$ bytes and $\epsilon = 0.1$. Further details on the workload generation are reported below for each experiment.

A. Experiment 1 (varying the utilization)

1) Workload generation: We tested systems with $m \in \{1, 4\}$ cores. For each utilization value $U \in \{0.05, 0.1, 0.15, \ldots, 0.95\}$ and TFFR $\in \{10^{-9}, 5 \cdot 10^{-9}, 10^{-8}\}$, we randomly generated m task sets, one per core as follows. One random processor was assigned utilization U, while the other $m - 1$ ones (if any) were assigned a utilization randomly selected in the range $[0.8 \cdot U, U]$. For each processor P_k, the individual task utilizations $U_{i,k}$ were generated with the Emberson et al.'s generator [13] and the WCETs $C_{i,k}$ were computed as $C_{i,k} = T_{i,k} \cdot U_{i,k}$ (given the periods generated as reported above). For each pair (U, TFFR), we generated 1000 systems and we tested each of them with Algorithm 1, keeping track of the ratio of the ones that resulted schedulable, i.e., those for which the algorithm manages the configure the memory test.

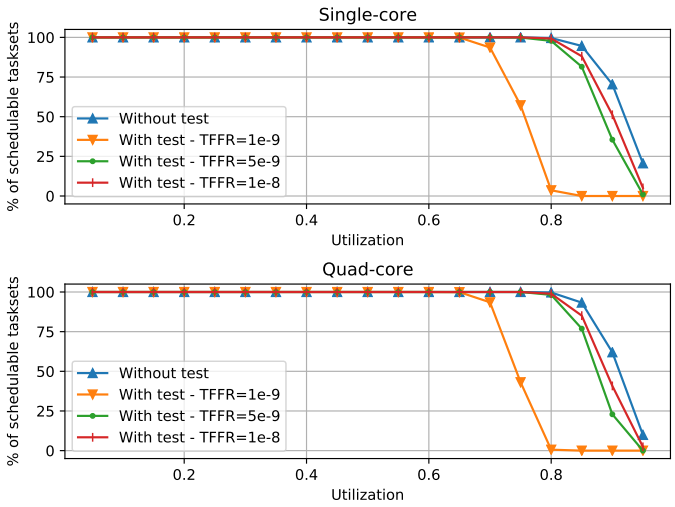

Fig. 8. Experiment 1: schedulability performance as a function of the system utilization U.

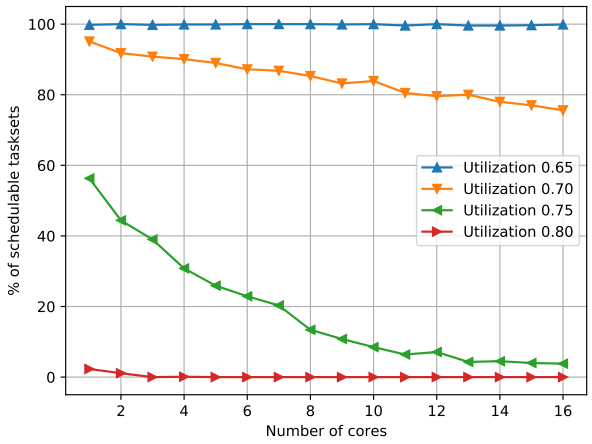

Fig. 9. Experiment 2: schedulability performance as a function of the number of cores.

2) Results: The results are reported in Fig. 8 along with the schedulability performance of the generated systems without integrating the memory test, which represents an upper bound of the achievable performance. For the figure it is possible to observe that the schedulability performance is much more sensitive to the TFFR rather than the number of cores m. Indeed, with TFFR $= 10^{-9}$ (corresponding to most stringent TFFR for SIL 4 systems), no systems can be schedulable when the utilization U exceeds 80%. Instead, with relaxed tolerable functional failure rates (TFFR $= 10^{-8}$, i.e., the least stringent TFFR for SIL 4 systems, and TFFR $= 5 \cdot 10^{-9}$), the schedulability performance becomes closer to the case without memory tests, hence suggesting that the impact of the tests becomes marginal. Note also that there is a little performance degradation passing from a single to a quad-core system, e.g., see the schedulability performance with utilization 80% for TFFR $= 10^{-9}$.

B. Experiment 2 (varying the number of cores)

1) Workload generation: For each utilization value $U \in \{0.65, 0.70, 0.75, 0.80\}$ and for each number of cores $m \in [1, 16]$, we generated the task sets as for Experiment 1 but keeping the per-core utilization in the range $[0.9 \cdot U, U]$ (to avoid perturbed results due to the variability of utilizations when the number of cores is large). For each pair (m, U) we tested 1000 systems as for Experiment 1 with TFFR $= 10^{-9}$.

2) Results: The results of this experiment are reported in Fig. 9 for four representative utilization values. From the figure it is clear that the number of cores that compose the system becomes a crucial parameter only for task sets with utilization falling in the critical range $[70\%, 75\%]$. Indeed for task sets with utilization lower than 70% and greater than 75%, the impact of the number of cores onto the schedulability ratio is negligible.

C. Experiment 3 (varying the test speed)

1) Workload generation: We tested with $m \in \{1, 2, 4\}$ cores. For each $\sigma \in \{0.8, 1.0, 1.2, \ldots, 2.6\}$, we generated the task sets as for Experiment 1. For each triplet (m, σ, U), we tested 1000 systems as described in Experiment 1 with TFFR $= 10^{-9}$.

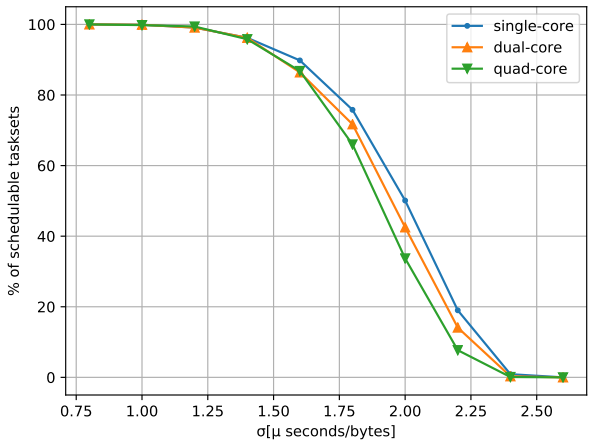

Fig. 10. Experiment 3: schedulability performance as a function of the test speed for $U = 0.75$.

2) Results: The parameter σ is related to the computational power of the system, i.e., it represents the maximum number of microseconds that a core takes to test one byte of memory. Fig. 10 reports the result for a representative utilization value ($U = 0.75$) and shows that the number of schedulable task sets quickly decreases for $\sigma > 1.75$. Furthermore, the figure shows the trend of the schedulability performance is quite similar for single-, dual-, and quad-core systems.

D. Experiment 4 (varying the TFFR)

1) Workload generation: We tested systems with $m \in \{1, 4\}$ cores, TFFR $\in [1{\cdot}10^{-9}, 6{\cdot}10^{-9}]$ with step $0.1{\cdot}10^{-9}$, and $U \in \{0.05, 0.1, 0.15, \ldots, 0.95\}$. Task sets have been generated as for Experiment 1 and, for each triplet (m, TFFR, U), we tested 500 systems with Algorithm 1.

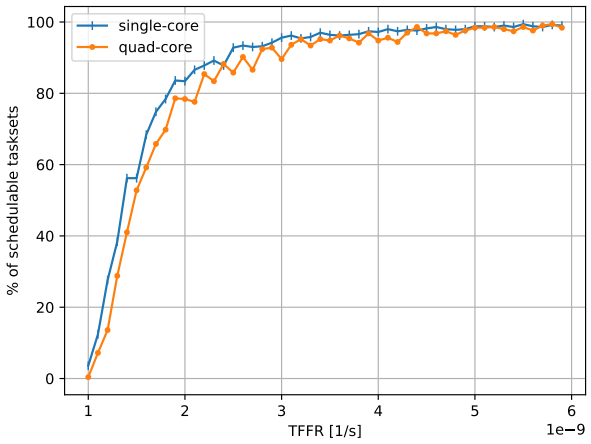

Fig. 11. Experiment 4: schedulability performance as a function of the Tolerable Functional Failure Rate (TFFR) for $U = 0.8$.

2) Results: The results of this experiment are reported in Fig. 11 for a representative utilization value ($U = 0.8$) and show that the schedulability ratio exponentially increases with the TFFR. This means that weaker safety constraints cause a big improvement in schedulable performance according to the solution proposed in this paper. Anyway, note that the proposed approach allows guaranteeing the schedulability of a system at the highest integrity level of the EN50129 safety standard, i.e., SIL 4, which requires $10^{-9} \leq \text{TFFR} < 10^{-8}$. Indeed, as it can be noted from the figure, a TFFR $= 4 \cdot 10^{-9}$ is already sufficient to achieve a very high schedulability performance for both single- and quad-core systems.

VII. Related work

Despite both academic and industrial researchers studied software-based self-tests (SBST) for memories during the last decades (e.g., see [14], [15], [16], [17]), such problems received limited attention from the real-time community, especially when considering multicore systems.

Moraes et al. [18] proposed a method to select a set of test routines from different test approaches to compose a test program for a single-core embedded platform. However, no schedulability analysis has been taken into account. Gizopoulos [19] presented a test selection algorithm that minimizes power consumption and the test execution time for a single-processor platform. The work aimed at obtaining the maximum fault coverage with a minimum impact on the systems' resources. The same author also investigated the integration of self-test routines in hard real-time uniprocessor systems, scheduling self-tests without affecting the deadline requirements of real-time tasks [20]. The approach was based on Rate-monotonic scheduling and a simple utilization-based test was used to ensure schedulability in the presence of the test task.

Paschalis and Gizopoulos [21] investigated the trade-off between fault detection latency and degradation of the system performance. They proposed a software-based self-test strategy that grouped the self-tests in a dedicated system process. This process might then be scheduled during idle periods by the operating system or at regular time intervals by using programmable timers.

Floridia et al. [22] tackled the integration of self-test routines with user application in a multi-core architecture for boot-time, self-test procedures. No real-time constraints have been considered. Reimann et al. [23] explored the integration of SBST at the system level considering the automotive domain. Overall, to the best of our records, none of the works in the literature fully address the challenge of integrating memory tests in a multicore real-time system while also considering response-time constraints as done in this work.

From the perspective of safety, researchers consolidated sever fault models and efficient functional tests for semiconductor memories. Functional safety standards of every safety-critical domain, as well as academic works such as [24], [25], [26], [27], provide extensive descriptions of the possible fault models and methodology for evaluating the fault coverage. Other research efforts have been devoted to intermittent faults [28].

VIII. Conclusion

This work proposed an approach to integrate software-based RAM tests in a real-time multicore system scheduled by fixed priorities. A software architecture has been presented to schedule and synchronize a test task on each core. The architecture also considers the replication of the test data and code to cope with the case in which the memory used by the test task itself has to be tested. Furthermore, by jointly considering a real-time task model and a safety model, this work proposed an algorithm to optimally configure a memory test while not violating the system schedulability and a safety requirement in the form of a TFFR bound. Experimental results showed that the impact of memory tests is not negligible when requiring stringent safety requirements (TFFR $= 10^{-9}$), but also that the schedulability performance tends to exhibit an exponential dependency on the TFFR. Indeed, in the tested cases, when selecting TFFR $> 4 \cdot 10^{-9}$ (that still allows guaranteeing the highest integrity level of the EN50129 standard) the proposed approach shows a minimal impact on the system schedulability.

Future work should investigate on methods to achieve a wider coverage of inter-word memory faults and techniques to support the online testing of other hardware resources.

References

[1] J. Henkel, L. Bauer, J. Becker, O. Bringmann, U. Brinkschulte, S. Chakraborty, M. Engel, R. Ernst, H. Härtig, L. Hedrich, A. Herkersdorf, R. Kapitza, D. Lohmann, P. Marwedel, M. Platzner, W. Rosenstiel, U. Schlichtmann, O. Spinczyk, M. Tahoori, J. Teich, N. When, and H.-J. Wunderlich, "Design and architectures for dependable embedded systems," in *2011 Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2011, pp. 69–78.

[2] "Cei en 50129," *Railway applications - Communication, signalling and processing systems - Safety related electronic systems for signalling*, 2019.

[3] "Cei en 60730-1," *Automatic electrical controls - Part1: General requirements*, 2019.

[4] S. Di Carlo and P. Prinetto, "Models in memory testing," in *Models in Hardware Testing*. Springer, 2010, pp. 157–185.

[5] A. J. Van De Goor, "Using march tests to test srams," *IEEE Design & Test of Computers*, vol. 10, no. 1, pp. 8–14, 1993.

[6] AUTOSAR, "Specification of ram test, autosar," 2017.

[7] S. Hamdioui, A. J. van de Goor, and M. Rodgers, "March ss: A test for all static simple ram faults," in *Proceedings of the 2002 IEEE International Workshop on Memory Technology, Design and Testing (MTDT2002)*. IEEE, 2002, pp. 95–100.

[8] P. Mazumder and K. Chakraborty, *Testing and testable design of high-density random-access memories*. Kluwer Academic Publishers, 1996.

[9] M. Bushnell and V. Agrawal, *Essentials of electronic testing for digital, memory and mixed-signal VLSI circuits*. Springer Science & Business Media, 2004, vol. 17.

[10] M. S. Abadir and H. K. Reghbati, "Functional testing of semiconductor random access memories," *ACM Computing Surveys (CSUR)*, vol. 15, no. 3, pp. 175–198, 1983.

[11] E. CEI, "Cei en 50126-1," *Railway Applications - The Specification and Demonstration of Reliability, Availability, Maintainability and Safety (RAMS). Part 1: Generic RAMS Process*, 2019.

[12] G. C. Buttazzo, *Hard real-time computing systems: predictable scheduling algorithms and applications*. Springer Science & Business Media, 2011, vol. 24.

[13] P. Emberson, R. Stafford, and R. I. Davis, "Techniques for the synthesis of multiprocessor tasksets," in *Proceedings 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2010)*, pp. 6–11.

[14] P. Bernardi, L. Bolzani, A. Manzone, M. Osella, M. Violante, and M. S. Reorda, "Software-based on-line test of communication peripherals in processor-based systems for automotive applications," in *Seventh International Workshop on Microprocessor Test and Verification (MTV'06)*. IEEE, 2006, pp. 3–8.

[15] 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*. IEEE, 2008, pp. 185–187.

[16] A. Benso, S. Di Carlo, and A. Savino, "Software-based self-test for reliable applications in railway systems," in *Railway Safety, Reliability, and Security: Technologies and Systems Engineering*. IGI Global, 2012, pp. 198–220.

[17] T.-W. Kuo, P.-C. Huang, Y.-H. Chang, C.-L. Ko, and C.-W. Hsueh, "An efficient fault detection algorithm for nand flash memory," *SIGAPP Appl. Comput. Rev.*, vol. 11, no. 2, p. 8–16, Mar. 2011. [Online]. Available: https://doi.org/10.1145/1964144.1964146

[18] M. Moraes, É. Cota, L. Carro, F. Wagner, and M. Lubaszewski, "A constraint-based solution for on-line testing of processors embedded in real-time applications," in *Proceedings of the 18th annual symposium on Integrated circuits and system design*, 2005, pp. 68–73.

[19] D. Gizopoulos, "Low-cost, on-line self-testing of processor cores based on embedded software routines," *Microelectronics journal*, vol. 35, no. 5, pp. 443–449, 2004.

[20] ——, "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, 2009.

[21] A. Paschalis and D. Gizopoulos, "Effective software-based self-test strategies for on-line periodic testing of embedded processors," *IEEE Transactions on Computer-aided design of integrated circuits and systems*, vol. 24, no. 1, pp. 88–99, 2004.

[22] A. Floridia, D. Piumatti, A. Ruospo, E. Sanchez, S. De Luca, and R. Martorana, "A decentralized scheduler for on-line self-test routines in multi-core automotive system-on-chips," in *2019 IEEE International Test Conference (ITC)*. IEEE, 2019, pp. 1–10.

[23] F. Reimann, M. Glaß, J. Teich, A. Cook, L. R. Gómez, D. Ull, H.-J. Wunderlich, P. Engelke, and U. Abelein, "Advanced diagnosis: Sbst and bist integration in automotive e/e architectures," in *Proceedings of the 51st Annual Design Automation Conference*, 2014, pp. 1–9.

[24] M. Riedel and J. Rajski, "Fault coverage analysis of ram test algorithms," in *Proceedings 13th IEEE VLSI Test Symposium*. IEEE, 1995, pp. 227–234.

[25] A. J. Van de Goor, I. Tlili, and S. Hamdioui, "Converting march tests for bit-oriented memories into tests for word-oriented memories," in *Proceedings. International Workshop on Memory Technology, Design and Testing (Cat. No. 98TB100236)*. IEEE, 1998, pp. 46–52.

[26] J.-F. Li, K.-L. Cheng, C.-T. Huang, and C.-W. Wu, "March-based ram diagnosis algorithms for stuck-at and coupling faults," in *Proceedings International Test Conference 2001 (Cat. No. 01CH37260)*. IEEE, 2001, pp. 758–767.

[27] A. J. van de Goor and J. De Neef, "Industrial evaluation of dram tests," in *Proceedings of the conference on Design, automation and test in Europe*, 1999, pp. 123–es.

[28] A. Lifa, P. Eles, Z. Peng, and V. Izosimov, "Hardware/software optimization of error detection implementation for real-time embedded systems," in *2010 IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2010, pp. 41–50.