IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2022.Doi Number

Improving the Efficiency of Software-Based Fault Protection Mechanisms with HUSTLE

N. Ferrante[1,2], L. Fanucci[1], F. Rossi[2], F. Terrosi[3], A. Bondavalli[2,3]

[1] Department of Information Engineering, University of Pisa, Pisa, PI, 56122 Italy
[2] Resiltech Srl, Pontedera, PI, 56025 Italy
[3] Department of Mathematics and Informatics, University of Florence, Florence, FI, 50134 Italy

Corresponding author: N. Ferrante (e-mail: nicola.ferrante@phd.unipi.it).

ABSTRACT To achieve confidence in safety-critical systems, requires among others to meet high requirements on online testing of computer systems, as dictated by safety standards such as ISO26262, IEC61508, and CENELEC EN 5012X. Online testing can be performed through the periodic execution of online SW Test Libraries, which are widely used in safety-related applications as a valuable safety mechanism to protect against random HW faults. SW Test Libraries introduce a non-negligible overhead on system performance, exacerbated by the increasing complexity of HW devices. This contrasts with the efforts of researchers and system designers for developing efficient systems. Reducing this overhead is an important achievement. We propose here HUSTLE, a Hardware Unit for SW-Test Libraries Efficient execution, which can be integrated into the chip design with minimum modification to the CPU's design. HUSTLE contains an Internal Memory, where the library code is stored, and sends instructions to the CPU, bypassing the Memory Subsystem. To further improve efficiency, we also propose a scheduling mechanism that allows to exploit the idle time of the CPU's execution unit. To show the efficiency gain in supporting the test libraries execution, we ran some experiments, where a considerable reduction of the overall CPU load was observed. Finally, remarks regarding the limited impact on the area and power consumption are presented.

INDEX TERMS Error Detection, Functional Testing, On-line Testing, Safety, SW-Test Libraries.

I. INTRODUCTION

Innovations in the field of Very Large-Scale Integration (VLSI) technologies and the advent of novel computing platforms have made the automation of complex tasks in constrained domains a reality [1]-[3]. The great potential of these innovations has led to an increasing interest in their adoption in many safety-critical application domains, such as the automotive, railway, and industry. These systems must fulfill the integrity requirements [4] set forth by standards developed by international committees, such as ISO[5], CENELEC[6], and IEC[7], with the aim of minimizing the risk of potentially catastrophic failures that damage human life and health. One of the main challenges in enabling the use of these technologies in safety-critical systems is the coexistence of three main characteristics: integrity, performance, and cost [8]-[10]. The lack of proper levels for one of these properties may lead to drawbacks that may prevent their adoption. Such challenges become more complex in situations in which a reduction in engineering costs and time-to-market is required. In such cases, valuable solutions must offer appropriate fault protection and mitigation mechanisms. Further, such mechanisms are required to be flexible and do not require heavy modification of the original design or excessively penalize its performance when applied in different contexts.

It is a general requirement from functional safety standards [5]-[7] to enrich the design of an embedded system with mechanisms (HW or SW or HW/SW) aiming to detect faults of the HW platform to improve its safe usage.

Fault-tolerance mechanisms can be based on both hardware (HW) and software (SW), each providing different levels of protection and targeting different failure modes [9] -[15]. HW-based techniques are faster but require either modifications to the original design or higher cost due to replication [14], [15], whereas SW-based techniques have no impact on HW cost but incur overheads that significantly reduce performance [12], [13], [16], [17].

Many SW-based mechanisms and mitigations have been proposed in the literature [8]-[13], [16], [17], such as defensive programming techniques, SW diversity, and purposely designed test routines.

In this context, SW-Test Libraries (STLs) are widely considered an effective mechanism to protect against permanent random HW faults [11], [16], [18]-[22]. STLs are sets of test routines providing high fault coverage and allowing compliance with well-established functional safety standards [23], [24].

To achieve high fault coverage, as required by the safety standards [5]-[7], STLs need to be scheduled with high frequency and this can negatively impact the performance of the embedded SW up to the extreme case to violate its timing constraints, then leading to a critical safety issue.

The solution proposed in this work is then introduced to counterbalance this problem enabling the proper usage of STLs on safety applications which require high computing resources, and, therefore, have an higher number of HW resources to be tested [17], [25]. This is the typical case of many SoCs used in ML applications for automotive where the embedded application cores (for example cluster of superscalar processors) are required to provide very high performance. This implies mainly two aspects: 1) the cores are not configured in lock-step mode not to lose computing resources and 2) an STL solution is then necessary to enable fault detection on the processors, then leading to the above challenge addressed in this paper.

Our proposal is called HUSTLE, a Hardware Unit for STL Efficient execution. It allows to i) host STL code in its internal memory and ii) provide STL instructions to the core without accessing the Memory Subsystem (MS). This way HUSTLE allows a reduction of the overhead imposed by the execution of the STL. Besides the basic mechanism, an additional benefit is brought by a mechanism that exploits architectural signals to detect the CPU execution unit's (from now on CPU for brevity) idle time and use this time to efficiently execute STL instructions.

This study provides a detailed description of the implementation of HUSTLE (extending preliminary concepts [26]) and offers an extended experimental campaign that accounts for the impact on the device area and its power consumption.

The remainder of this paper is organized as follows. Section II provides the background, Section III describes the implementation of HUSTLE, Section IV presents the details of the experimental campaign, Section V discusses the results, Section VI provides a post-synthesis evaluation of the impact on device area and power consumption, and Section VII reviews the related works found in the literature. Finally, Section VIII concludes this paper.

II. BACKGROUND

Safety-critical systems must achieve stringent dependability and integrity requirements, imposing constraints on their design from both hardware and software viewpoints.

To maintain the target integrity level, it is necessary to implement protection techniques such as STLs, which must run periodically to monitor the integrity of the system. The execution of STL must interleave with the execution of the functional code (payload).

To determine the execution period of an STL the system designer has to know the required Fault Tolerant Time Interval (FTTI) defined as the "minimum time span from the occurrence of a fault in an item to a possible occurrence of a hazardous event, if the safety mechanisms are not activated" [5]. Knowing the FTTI, the system designer must define the STL execution period such that faults are detected and handled in a time interval lower than the FTTI. This time interval is also called the Fault Handling Time Interval (FHTI), and is composed of two parts: the time necessary to detect a fault, that is, the Fault Detection Time Interval (FDTI), and the time necessary to *react* to the occurrence of a fault, that is, the Fault Reaction Time Interval (FRTI). In Fig. 1 illustrates a schematic view of these quantities in relation to the STL scheduling period.

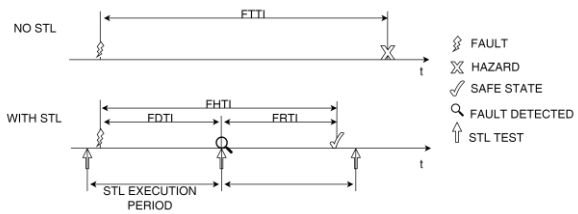

Fig. 1 **Schematic representation of the FTTI on the upper, and of the FHTI, decomposed in FDTI and FRTI in the lower part of the figure.**

This continuous interleaving between the payload SW and STL causes a non-negligible overhead, thereby reducing the system performance. To offer a simple example, suppose that there are two tasks: payload $task_A$, which requires t_A time units to complete, and STL $task_{STL}$, which requires t_{STL} time units to complete. To guarantee the correctness of the system, $task_{STL}$ must be run before each execution of $task_A$. To precisely define and characterize the overhead incurred in this execution, we assume that interrupts to be disabled as serving an interruption cannot be classified as overheads. Under this assumption, the total time required for one complete execution of the task set is $t_e = t_A + t_{STL} + t_o$, where t_o is the time spent by the hardware to handle asynchronous events such as cache misses and mispredictions.

In the computation of t_o we consider the effects of many HW events, such as mispredictions, pipeline stalls, and cache misses, which can impact the execution time of the task set. The overhead introduced by the system scheduler to handle the execution of multiple tasks is not included in t_o but is considered part of t_A and t_{STL}.

Fig. 2 shows an example of a typical system run. In the ideal case, $t_o = 0$ as shown in the upper part of Fig. 2. However, in a real execution, owing to asynchronous events such as cache misses or branch mispredictions, there are times in which the CPU is idle, as can be seen from the lower part of Fig. 2.

In reality $t_o > 0$. In fact, CPUs are likely to be idle, waiting for instructions from the memory subsystem, for example, because of instruction cache misses. The amount of overhead introduced by each cache miss is variable and depends on the location of the instructions in the memory hierarchy.

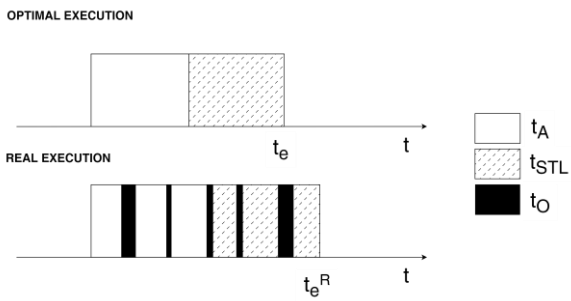

Fig. 2 **Scheduling of the two tasks. The white area represents the execution of taskA, dotted area represents taskSTL, while the black area the overhead due to hardware-specific events**.

Taking as a reference system the one depicted in Fig. 3, the instruction may be located in the L1 instruction cache, L2 cache, or main memory. The higher the level of the hierarchy that needs to be traversed, the larger the amount of time required to retrieve the instruction. Moreover, some resources are shared between the components of the system; for example, when two cores need to access the L2 cache at the same instant, they must compete to communicate with the L2 cache. Consequently, the overhead for retrieving the instruction increases.

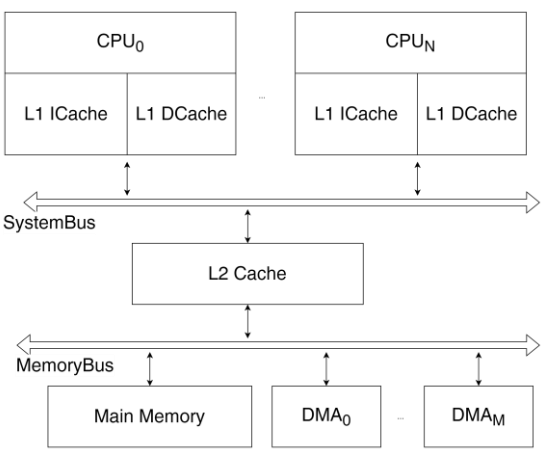

Fig. 3 **Block Diagram representing a generic system composed by N CPUs, with private instruction and data L1 caches. A shared L2 Cache, a main memory and DMA devices.**

The problem of retrieving instructions from lower levels of the memory hierarchy arises both for payload and STL execution. In this study, we attempted to eliminate or reduce t_o. This was performed in two steps. First, we provide a solution to ensure that no overhead is incurred while executing the STL. Then, we attempt to reduce the portion of overhead incurred during payload execution.

III. HUSTLE

In this Section, we provide a detailed description of how the problem of the overhead on execution due to retrieving instructions from the memory hierarchy has been addressed by leveraging HUSTLE. In Section III.A, we address the problem of reducing the overhead incurred during the execution of STL code. In Section III.B, we provide a description of an enhancement that enables HUSTLE to automatically handle the scheduling of STL tests. In Section III.C, we describe how, by exploiting an efficient scheduling mechanism and architectural enhancements to the basic architecture, we can also reduce the overhead experienced during payload execution.

We followed two design principles for HUSTLE, which can be summarized by the following requirements:

1) The solution provided shall not modify the internal structure of the CPU
2) The solution provided shall not impose constraints on the STL implementation.

These two design principles allowed us to have the minimum possible impact on the device area with a low effort for the integration of HUSTLE in different HW architectures, while allowing it to be used with different STL implementations.

A. HUSTLE BASIC ARCHITECTURE

To avoid cache misses during STL execution, we propose the architecture described in Fig. 4.

HUSTLE was placed between the CPU and the Memory Subsystem (MS). Internally, it has an Internal Memory (IM) and a ByPass Logic (BPL). The IM is used to store STL instructions, whereas the BPL orchestrates the communication between three elements: the CPU, which requests instructions from the memory; the MS, which handles requests from the CPU for functional code; and HUSTLE's IM, which handles requests related to STL code. The BPL is completely transparent to the core because it acts as a simple switch that does not introduce any delay in communication between the CPU and MS.

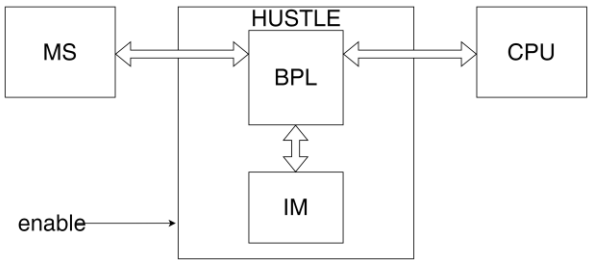

Fig. 4 **HUSTLE's Block schema**

This implementation provides a fast and independent channel to send STL code to the core as it relieves the Memory Subsystem (MS) from handling STL instructions, as depicted in Fig. 5, and fulfills our requirements because it is placed outside the core and does not impose any constraint on the design of the STL code.

Indeed, since the proposed architecture is intended to be used in Safety Critical systems, it is fundamental to consider potential hazards impacting on systems' security introduced by HUSTLE. Surely, if an attacker were able to access the IM, and manipulate its content arbitrarily, this would cause severe security threats. Fortunately, traditional solutions for tackling this kind of issue are applicable to HUSTLE, since it is not different from any other memory area of the system. Thus, one option to secure the IM could be leveraging the CPU memory protection unit, marking this area as non-writeable. Moreover, in high-criticality applications, another solution is to implement the IM as a ROM memory, which cannot be programmed at runtime. Finally considering also the possibility that an attacker can gain physical access to the system, and compromise the IM by breaking the boot process and ROM programming procedure, additional mechanisms to authenticate the content of the IM, based for instance on Hashed Message Authentication Codes (HMAC) can be implemented in the HUSTLE logic, hardcoding a secure key within it, making unfeasible for an attacker to arbitrarily modify the content of HUSTLE IM.

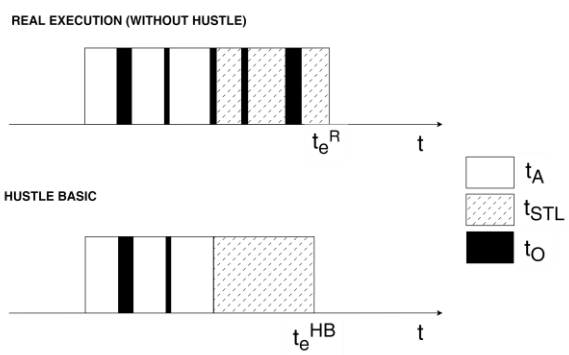

Fig. 5 **Comparison of two executions: one without HUSTLE in the upper and one with HUSTLE in the lower part of the figure.**

To describe the behavior of HUSTLE during system operation, a Finite State Machine (FSM), represented in Fig. 6 is provided. It can be observed that it is composed of three states:

OFF: In this state, HUSTLE is disabled, and the BPL is completely transparent: the request and response signals between the CPU and MS pass unmodified through HUSTLE.

IDLE: In this state, HUSTLE forwards requests and responses related to the functional code from the CPU to the MS and vice versa. (this state corresponds to the execution of the payload code)

OPERATIONAL: In this state, the BPL handles requests and responses related to non-functional code from the CPU to the IM and vice versa. (this state corresponds to the execution of the STL code)

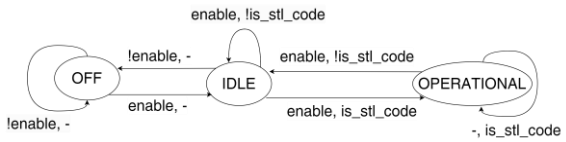

Fig. 6 **Finite State Machine showing the states and the transitions of HUSTLE module. The arrows are labeled with signals that enable the firing the state transitions.**

This automaton has 2 input signals to drive transitions:

enable: If enable is asserted HUSTLE goes into the IDLE state becoming active.

is_stl_code: If the is_stl_code signal is asserted when HUSTLE is in the IDLE state, HUSTLE transitions into the *OPERATIONAL* state. The BPL generates this signal by checking the address of the instructions requested by the core during the execution. In this basic architecture, in order to be able to periodically execute the STL, the system designer must allocate an HW timer or rely on the system scheduler. Very often, in order to meet the system scheduling constraints (on the payload SW), the STL cannot be executed all at once, but the execution needs to be split into several parts. Moreover, some of the tests included in an STL cannot be interrupted, therefore careful scheduling of the STL 'pieces' has to be defined. Having defined such system-level scheduling, HUSTLE's role is to respond to CPU requests for the STL code whenever the scheduler decides to execute parts of the STL. We refer to this method of using HUSTLE as *Passive* mode.

B. HUSTLE'S ENHANCED ARCHITECTURE

HUSTLE allows also a completely different system organization: while in Passive mode the system and the scheduler have visibility of Payload tasks and of the STL task (which resides in the HUSTLE memory) a new '*Active*

mode' is possible whereby the entire management and scheduling of the STL is performed within HUSTLE and the system becomes unaware of the existence of an STL. In this mode, HUSTLE manages the scheduling of STL pieces by issuing interrupt requests to the CPU. This is achieved, as shown in Fig. 7, by adding two further elements: Interrupt Generation Logic (IGL) and Test Scheduling Logic (TSL).

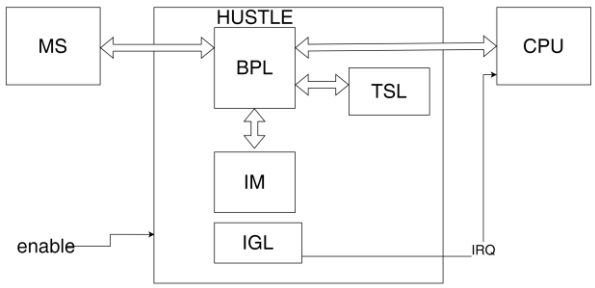

Fig. 7 **Enhanced HUSTLE Architecture, the Test Selection Logic (TSL) is connected to the BPL. The IGL generates interrupt requests to the CPU with an IRQ signal.**

The Interrupt Generation Logic (IGL), which features an internal timer, needs to be configured by the system designer in accordance with the required execution period of the STL. The Test Scheduling Logic (TSL) is designed and implemented to automatically handle the selection of the next STL fragment to execute. In particular, when the IGL timer expires, the IGL generates an interrupt request asserting a dedicated signal that is routed to the CPU interrupt controller (IRQ). When the core handles the interrupt request and jumps to the STL code, HUSTLE provides instructions to the CPU and the BPL asks the TSL the address of the next portion of the STL to run. When using this execution mode, the system designer only needs to provide an Interrupt Service Routine (ISR) to handle the interrupt generated by HUSTLE and jump to the entry point of the STL, then HUSTLE will then automatically handle the execution, providing the core with the appropriate instructions.

Thus, with this enhanced architecture, we have made the execution of STL almost transparent to the rest of the system, providing a mechanism that autonomously handles its execution.

We want to highlight that when using the *Active* configuration, it is important to carefully handle interrupt generation and prioritization to maintain the schedulability of the task set. In the presence of such interrupts, tasks can be considered aperiodic by the scheduler. However, consolidated solutions exist in the state-of-the-art for the scheduling of aperiodic tasks [27]-[29] hence, we argue that it is feasible to schedule a payload task set given that a proper analysis is performed. In this study, we did not elaborate on such aspects further.

C. HUSTLE EFFICIENT SCHEDULING

The HUSTLE's enhanced architecture allows to alleviate the overhead due to cache misses or memory access during the execution of the payload SW by keeping the CPU busy executing some carefully selected fragments of the STL code during the time the CPU would otherwise wait for payload instructions to be retrieved from the MS. If we can provide STL instructions to the CPU sufficiently fast, without interfering with the MS, while retrieving the payload instructions, we can optimize the usage of the CPU. Consider the situation represented in Fig. 8, where a cache miss occurs during the execution of $task_A$. The CPU handles the cache miss, leaving the core idle for some time and incrementing the overall execution time. By exploiting this time executing a portion of the STL, we can avoid this overhead.

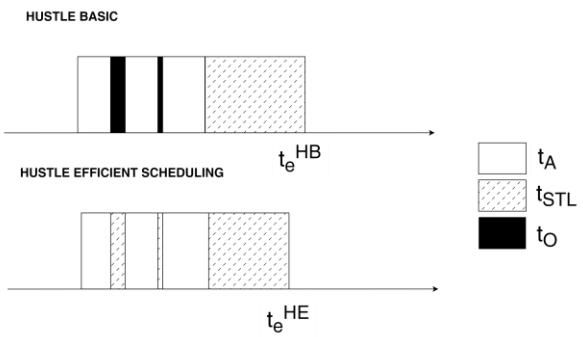

Fig. 8 **Comparison of the execution with the HUSTLE enhanced architecture, upper, and HUSTLE efficient scheduling, lower part of the figure.**

To enable this mechanism, we routed the cache miss signal to HUSTLE's IGL, slightly modifying the IGL to generate an interrupt request when a cache miss occurs.

This improved HUSTLE architecture generates interrupts according to two different modes:

Periodic: The interrupt is generated periodically by using the internal timer.

Cache Miss Driven: The interrupt is generated in correspondence of a cache miss.

It is important to note that a possible issue that may arise when using this mechanism is the unpredictable length of the cache miss resolution time. Indeed, cache misses require a different amount of time to be resolved depending on the miss occurring in the L1 or L2 cache. This may impact the benefits provided by this scheduling mechanism. STL fragments longer than the cache miss resolution time would allow the processor to be kept busy all the time, whereas STL fragments that are shorter than the cache miss resolution time imply some idle cycle for the processor, but would leave the execution time of the payload untouched.

Indeed, in an application when most cache misses happen in the high-level cache our mechanism offers partial benefits, However, considering complex payload SW that cannot

entirely fit into the cache memory, it is not rare that the code needs to be retrieved directly from the DRAM, causing relatively long idle times due to misses in the last level of cache.

In the experimental campaign described in Section IV, we demonstrated that it is possible to execute relevant portions of the STL code without impairing the payload response time.

D. HUSTLE INTEGRATION

To validate HUSTLE, its enhanced architecture, and its mechanism to efficiently exploit the idle time of the CPU to execute STL instructions, we integrated our solution into a complete System on Chip (SoC). We selected the Rocket Chip[30], made available by the Chipyard framework [31]. The framework provides facilities for building a customizable SoC, including the possibility of choosing between different RISC-V [32] CPUs architectures. In this work, we selected as the target CPU architecture the Berkley Out-of-Order Machine (BOOM) Core [33]-[35] a superscalar, highly configurable out-of-order application-level CPU.

HUSTLE was placed between the BoomCore and the L1 Instruction Cache, as shown in Fig. 9. The IM size was configured to 32KB which was enough to host the STLs used in our experimental campaign.

We selected the *SmallBOOM* implementation of BoomCore, a single pipelined core, TABLE I reports some of the main parameters of this implementation.

To configure the behavior of HUSTLE at runtime, such as enabling or disabling the module, selecting the execution mode, and other functionalities described hereafter, we added some Control and Status Registers (CSR) to the core:

HUSTLE_CTRL: Provides basic control functionalities, such as enable/disable, execution mode selection, and interrupt generation configuration selection.

HUSTLE_BASE_ADDRESS: Used to set the address of the entry point of the STL.

HUSTLE_TEST_PERIOD: Used to set the period in clock cycles for the execution of STL fragments when HUSTLE is used in *Active* mode with the *Periodic* interrupt generation configuration.

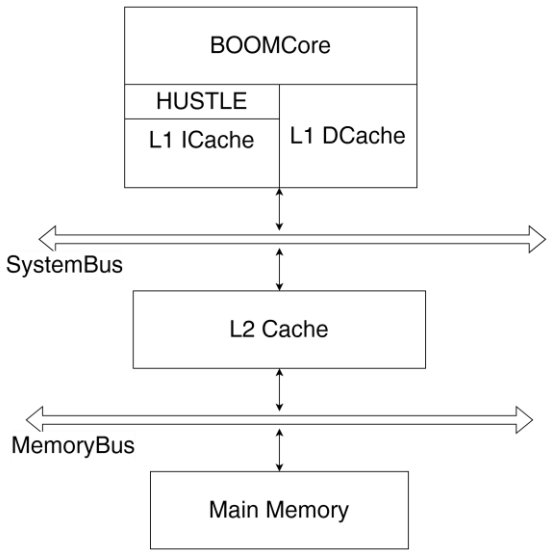

Fig. 9 **Rocket Chip Architecture used in the experimental campaign. The HUSTLE module is integrated between the BOOMCore and the L1 ICache. A system bus is used to connect the L2 Cache and the Core. The L2 Cache is connected to the Main Memory using a Memory Bus.**

The addition of these CSRs slightly increased the size of the register file. However, this simple solution is not the only possible solution; other solutions exist, such as memory-mapped registers, which are less intrusive in core design. The rationale behind this design choice is dictated exclusively by the simplicity of its implementation within the Chipyard Framework.

TABLE I
BOOM CORE CONFIGURATION

Core width	1
Fetch width (Bytes)	8
ICache size (KB)	16
DCache size (KB)	16
L2 Cache size (KB)	128

Boom Core Configuration Parameters. In this table only a restricted set of parameters is reported. The complete list of parameters can be found in [35]

IV. EXPERIMENTAL CAMPAIGN

In this section, we list the technical details of our experimental campaign. In our experimental campaign, the aim was to assess the performance benefit obtained using HUSTLE by comparing a solution without HUSTLE with one with it (in both Periodic and Cache Miss-Driven Scheduling). We performed such an evaluation on the target architectures described in Section IV.A while executing the test suite described in Section IV.B. Section IV.C describes the experimental setup. Finally, the evaluation metrics are defined in Section IV.D.

A. TARGET SYSTEM ARCHITECTURE

The experimental campaign was executed using three different configurations of the system architecture. These three architectures were chosen to allow us to observe the effect that an increase in the cache miss resolution time may have on the execution, and how this impacts the benefits provided by HUSTLE. To this extent, we modified the configuration of the rocket-chip to introduce interference at two different points of the MS, the L2 Cache, and the Main Memory.

The first architecture was a single-core architecture, from now on, *SingleCore*. In this architecture, the core is the only device that sends requests to the MS.

The second architecture has two identical CPUs, we will refer to this architecture in the next sections as *DualCore*. Its purpose is to allow the investigation of the effect that the contention may have in systems with an MS shared between multiple users, such as, CPUs and DMA devices. This architecture allows observation of the effect of contention on the L2 Cache caused by the simultaneous requests of the two cores to the MS.

Finally, the third architecture has a slower Main Memory. To this extent, we reduced the MemoryBus frequency from 1000 to 500 MHz. We refer to this architecture as the *SlowMem* architecture.

TABLE II reports the details of the three configurations used in the experimental campaign.

TABLE II
ROCKET-CHIP ARCHITECTURES

	Architecture		
	SingleCore	*DualCore*	*SlowMem*
CPU Frequency	*1.6 GHz*	*1.6 GHz*	*1.6 GHz*
Number of CPUs	*1*	*2*	*1*
MemoryBus Frequency	*1000MHz*	*1000MHz*	*500MHz*

B. TARGET SOFTWARE

The SW stack used for the execution of the tests is composed of:

- A payload SW, composed of three tasks, namely *taskA*, *taskB*, and *taskC*, and an *idle* task. The idle task is executed after the three tasks when the core waits for the expiration of the scheduling period.
  The three tasks execute the same code; however, the code of each task was placed in a different memory region. The SW executed by the three tasks is an enhanced version of the *fillCache* SW used in [26], which was enhanced to fill the L2 Cache.
  It executes a sequence of heterogeneous instructions, such as arithmetic operations (integer and floating-point), and memory accesses. These instructions are sequentially placed in memory. The *fillCache* SW allows the MS to be stressed, generating a large number of cache misses, both in the L1 Instruction Cache and L2 Cache. This was chosen to reproduce a worst-case execution scenario for application cores.
- A minimal OS, composed of a boot sequence that performs startup operations and initialization of the rocket chip, a set of routines used to configure HUSTLE and handle interrupt requests, and a cyclic executive scheduler that executes a fixed number of iterations. The number of iterations is configurable at compilation time.
- Three different STL implementations. Each STL is composed of several STL fragments, implemented in assembly code, that stimulate different modules of the CPU. STL fragments are sets of test routines executed sequentially. The STL is provided with a scheduling API implemented in C to allow the user to select and execute the desired STL fragment. The three STLs contain the same test routines but differ in how they are grouped within different STL fragments.

C. SETTING OF THE EXPERIMENTS

Hereafter, we provide a detailed description of the settings used in the experimental campaign. The experiments were performed first to understand how many STL instructions can be executed while a cache miss is resolved and then to compare the overhead reduction observed for the three reference architectures.

To understand how many STL instructions can be executed while a cache miss is resolved, we use the HUSTLE configuration driven by cache misses, as described in Section III.C. We focus on a specific cache miss and then vary the size of the executed STL fragment. This phase of the experimental campaign was performed only on the *SingleCore* architecture, as the validity of the results can be easily extended to other architectures.

We executed the three tasks by measuring the time necessary to resolve each cache miss that occurred. The Cache Miss Resolution time is computed as the difference between the time when the cache miss occurs and the time when the L1 cache makes the instruction available to the Instruction Fetch Unit of the CPU. We then select a cache miss that requires a sufficiently long time to be resolved. We then exploit HUSTLE: when the selected cache miss occurs,

we issue an interrupt request to inject a sequence of instructions during the cache miss resolution time.

In particular, we selected the 88th cache miss with a resolution time of 320 cycles and repeated the test with an increasing number of instructions, from 50 to 300 in steps of 50, measuring 1) the number of additional instructions executed and 2) the number of additional clock cycles necessary to complete the execution. The parameter settings for these experiments are reported in TABLE III.

TABLE III
PARAMETERS SETTINGS FOR THE FIRST PHASE OF EXPERIMENTAL CAMPAIGN

Average number of Instructions Per Task Execution	2584
Selected cache miss resolution time	320 clock cycles
Number of STL instructions injected	[50, 300] Step of 50

In the main experiment, we evaluated different combinations of HUSTLE configurations by executing the test SW suite described in Section IV.B on the different HW architectures presented in Section IV.A. The test performed in this phase aimed to evaluate the HUSTLE execution modes for the STL described in Section III. Before entering into the details of the setting of the experiments, we need to define two fundamental parameters: the Safety Period and the STL Scheduling Mode.

Safety Period (SP): SP denotes the STL scheduling period. To evaluate the impact of this parameter, we selected two SP: 10ms, and 1ms. Recall that SP is not a free parameter that can be chosen arbitrarily but needs to be derived through an accurate safety analysis and is strongly application dependent. Here, the SP values were selected considering the reasonable application requirements for state-of-the-art automotive applications.

Note that SP is the time interval in which the STL must be *entirely* executed to guarantee its nominal protection level. The fragments composing the STL can then be executed all at once in a single invocation or split into many parts whose execution is spread throughout the SP.

STL Scheduling Mode (SM): To explore the different possibilities enabled by HUSTLE to schedule STL, we chose the following scheduling modes:

- *Standard (STD)*: the STL is executed periodically, without using HUSTLE.
- *HUSTLE_Periodic (H_P)*: HUSTLE is in Active execution mode, in Periodic Configuration, and
- *HUSTLE_Cache_Miss_Driven (H_{CM})*: HUSTLE is in Active execution mode in the Cache-Miss-Driven configuration.

Note that in STD, the C API provided by the STL is used for the selection of STL Tests to execute, whereas in H_P and H_{CM} this is accomplished by HUSTLE's TSL.

TABLE IV reports the parameter settings for this phase of the experimental campaign. Note that in the DualCore configuration, the first core executes the payload SW, whereas the second one executes a workload composed of a single task which perform continuously read operations, to ensure that L2 cache interference between the two cores is properly activated.

Finally, we provide a comparison of HUSTLE with an alternative approach, that is using a larger instruction cache. In this experiment we modified the *SingleCore* architecture, by removing HUSTLE and increasing the IC size to 32KB and 64KB.

The experiments were executed with an RTL simulator of the Rocket Chip. Each simulation lasts for 10ms of simulated time, after which a timeout expired and stopped the simulation. The configuration of the SoC is the default configuration provided in the Chipyard Framework. Synopsys VCS was used to compile the RTL.

TABLE IV
MAIN EXPERIMENT PARAMETERS SETTINGS

Payload Scheduling Period	0.043ms
Average number of Instructions Per Task Execution	2584
Safety Period	1ms, 10ms
STL Fragment Size (number of instructions)	STL_1: 850, STL_2: 85, STL_3: 50
Number of STL Fragments	STL_1: 40 STL_2: 400 STL_3: 680
C API Size (average number of Instructions)	1614

D. EVALUATION METRICS

HUSTLE was evaluated by executing a test suite and observing how CPU utilization was affected by the usage of different execution modes in different execution scenarios. The results obtained from these experiments were compared with an execution with standard STL scheduling (without HUSTLE).

Hereafter, we report the definition of the metrics used during our experimental campaign:

*Response Time Increase (*ΔRT*)*: This metric is used in the

first phase of the experimental campaign and is computed as the difference between the time *t* necessary to execute the workload, and time spent to execute the same workload plus the STL code (t_{STL}). Thus $\Delta RT = t_{STL} - t$.

CPU Load (U): This metric is used in the main experiment, and it is computed as the time *t* spent by the CPU in executing its workload, that is, the task set, OS, and STL, divided by the total execution time t_{TOT}. Thus $U = t/t_{TOT}$.

V. RESULTS

In this section, we discuss the results of our experimental campaign.

A. STL EXECUTION DURING A CACHE MISS

The results obtained are reported in Fig. 10 and highlight that if a cache miss takes enough time to be resolved, a considerable number of instructions can be executed while the core is waiting for instruction retrieval from the main memory. The figure shows that, in a cache miss resolution time of 320 clock cycles, a stream of up to 200 instructions can be injected and executed without any increase in the response time.

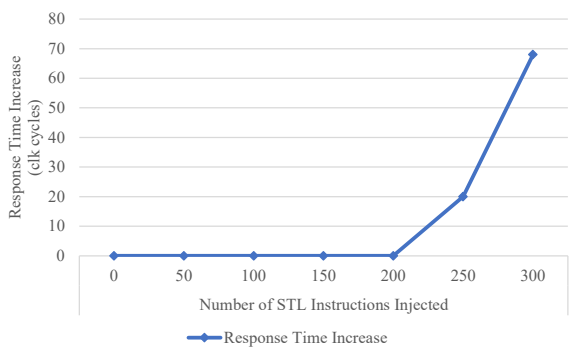

Fig. 10 **Results of the first phase of the experimental campaign, the line represents the response time increase varying the number of STL instructions injected.**

Consider that 20 instructions are needed by the ISR to handle the interrupt request, and additional clock cycles are wasted because of the pipeline flush that occurs when an interrupt request is served and after the return from the ISR. However, a large part of the cache miss resolution time can be profitably used for the STL code. This is a remarkable result showing that exploiting HUSTLE cache miss-driven configuration allows the execution of significant fragments of the STL code without any increase in the response time of the payload SW.

B. MAIN EXPERIMENT

The main experimental campaign has the objective of observing the impact on the CPU Load of using different Scheduling Modes provided by HUSTLE in combination with different Safety Periods, comparing the results with an execution without HUSTLE.

SingleCore

The results of the simulation campaign for the *SingleCore* architecture are reported in TABLE V. We can see that the CPU Load (U) always decreases when HUSTLE is used, this is expected since, when using HUSTLE, the overhead of the C API is removed and the STL Test scheduling does not require additional instructions, since it is accomplished automatically by HUSTLE. Moreover, instruction retrieval from HUSTLE's Internal Memory does not suffer from cache misses. We can see that by changing the SP, the CPU Load increases significantly when using the STD scheduling mode. Additionally, increasing the FS may lead to the impossibility of scheduling the task set together with the STL. Conversely, this problem never arises when HUSTLE is enabled.

With a 1ms SP, the STD scheduling strategy can still be used with the default configuration, but does not support a reduced Fragment Size. HUSTLE allows to save more than 4% of the CPU Load in the default fragment size and allows to run easily shorter fragments. The lowest values of CPU load are concentrated in the row corresponding to the H_{CM} scheduling mode.

Note that when the Fragment Size decreases, the CPU load increases in the STD and H_P scheduling modes, whereas it remains essentially constant in the H_{CM} scheduling mode; in particular, with a 1ms SP and a Fragment Size of 50 instructions, a decrease in CPU Load is observed. This is expected because smaller Fragments of the STL are more likely to fit the execution during the time span when payload instructions are fetched from the MS.

Because we do not perform any selection of the cache misses, some overhead is expected owing to some cache misses with a fast resolution time (observed in the order of 20 clock cycles for L1 cache misses that hit L2 cache).

TABLE V
CPU LOAD (U) ON SINGLECORE ARCHITECTURE

Baseline *85.67%*						
SP	10ms			1ms		
Fragment Size	850	85	50	850	85	50
STD	*86.35%*	*90.39%*	*93.44%*	*92.33%*	*OL*	*OL*
H_P	*85.87%*	*86.18%*	*86.36%*	*88.26%*	*90.58%*	*92.13%*
H_{CM}	*85.89%*	*85.89%*	*85.92%*	*88.01%*	*88.02%*	*87.90%*

Measurements of CPU load (U) from the test campaign performed on the *SingleCore* architecture, OL means, OverLoad, i.e., the CPU load is above 100%.

DualCore

The results of the simulation for the *DualCore* architecture are presented in TABLE VI. We can notice that the contention on the L2 Cache, caused by the addition of one core, brings a substantial degradation of the system performance, which can already be observed in the baseline execution, where an increase in the CPU load of 5.69% is observed. Nevertheless, the results of the experiments were consistent with those observed in the previous case. Only the contention on the L2 Cache experienced in this architecture results in one additional case in which the CPU is overloaded.

TABLE VI
CPU LOAD (U) ON DUALCORE ARCHITECTURE

Baseline *91.36%*						
SP	10ms			1ms		
Fragment Size	850	85	50	850	85	50
STD	*92.08%*	*96.05%*	*99.15%*	*98.00%*	*OL*	*OL*
H_P	*91.56%*	*91.90%*	*92.10%*	*94.04%*	*96.31%*	*OL*
H_{CM}	*91.63%*	*91.65%*	*91.74%*	*93.63%*	*93.52%*	*93.47%*

Measurements of CPU load (U) from the test campaign performed on the *DualCore* architecture, OL means, OverLoad, i.e., the CPU load is above 100%.

Moreover, focusing on the results for an SP of 10ms, in the experiment with a Fragment Size of 850 instructions, the H_{CM} configuration showed slightly worse results than the H_P configuration. Analyzing the simulation results, we found that this fluctuation is due to the effects of L2 cache contention, causing an increase in the execution time, mostly concentrated on OS routines; however, the CPU load is still lower than in the STD scheduling mode. With SP of 1ms the H_{CM} shows a decreasing CPU load for a smaller fragment size.

SlowMem

The results of the simulation for the *SlowMem* architecture are presented in TABLE VII. In this architecture, the performance degradation was the most significant of the experimental campaign. The number of cases where CPU overload occurred increased to 7 out of 18. Here, we can observe that when the SP is 1ms, standard scheduling always fails, whereas when using HUSTLE, we are still able to maintain some margin.

TABLE VII
CPU LOAD (U) ON SLOWMEM ARCHITECTURE

Baseline *96.19%*						
SP	10ms			1ms		
Fragment Size	850	85	50	850	85	50
STD	*96.94%*	*OL*	*OL*	*OL*	*OL*	*OL*
H_P	*96.43%*	*96.64%*	*96.93%*	*98.76%*	*OL*	*OL*
H_{CM}	*96.44%*	*96.34%*	*96.33%*	*98.44%*	*98.12%*	*97.71%*

Measurements of CPU load (U) from the test campaign performed on the *SlowMem* architecture, OL means, OverLoad, i.e., the CPU load is above 100%.

In this architecture, the H_{CM} configuration shows a decreasing CPU load at a lower fragment size for both the SP cases. This may depend on the fact that, in this configuration, cache misses take, on average, more time to be resolved. This allows to fit in this time larger portions of the STL fragment execution. In Fig. 11, the average cache miss resolution time is reported for the three architectures. This was measured on the baseline execution, that is, that with only the payload SW in execution.

Consistently with the experiments performed when the cache miss resolution time is higher, HUSTLE with cache miss-driven scheduling combined with a small STL Fragment Size is more beneficial.

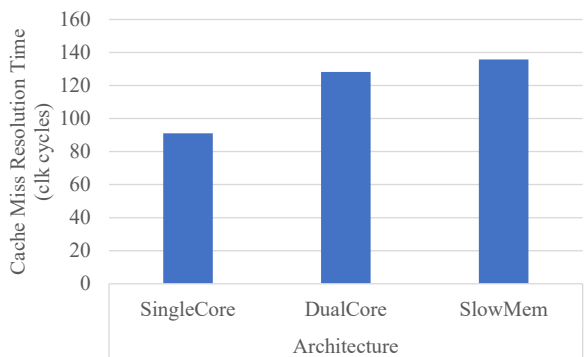

Fig. 11 **Average cache miss resolution time in the different architectures.**

C. COMPARISON WITH LARGER CACHE SIZE

An additional study compares HUSTLE with different CPU implementations with an increased L1 instruction cache size (32KB and 64KB). This is done to consider the potential benefit of using the resources of the HUSTLE internal memory to increase the L1 cache size instead of having it as part of additional HW adopted for HUSTLE. In this context

we executed the STL with the STD scheduling mode. The results of the experiments are reported in TABLE VIII. As can be noted. Increasing the IC size reduces the baseline execution time. However, this HW configuration performs better than HUSTLE just in one experiment, the one with largest cache (64KB), and FS of 850 instructions. This happens because of a reduction of the baseline execution time. However, when reducing the FS or decreasing the SP, HUSTLE performs better in every other experiment. This is expected since, differently from cache memory, the IM provides an independent channel, which hosts the STL code without replacing it.

VI. POST-SYNTHESYS ASSESSMENT

In addition to evaluating the performance benefits of HUSTLE through the experimental campaign described above, we also evaluated its impact in terms of overhead on the chip area and power consumption.

To assess the overhead on HW area, we synthesized the two variants of the *Small* CPU implementation, with and without HUSTLE, on FPGA using Xilinx Vivado 2022.2. The target device selected is the UltraScale+ MPSoC, in particular the xczu7ev device. The selected target clock frequency is 100MHz.

The overhead on power consumption was evaluated using vector-less power analysis, by varying the input toggle rates. The analysis was performed using the Power Report tool, integrated into Xilinx Vivado 2022.2. The measurements of power consumption are taken in four different situations: 1) when the CPU is not equipped with HUSTLE, 2) when the CPU is equipped with HUSTLE and the module is in OFF state, 3) when the CPU is equipped with HUSTLE and the module is in IDLE state, and 4) when the CPU is equipped with HUSTLE and the module is in OPERATIONAL state, by setting the signals shown in Fig. 6 accordingly.

The metrics used for this post-synthesis evaluation are as follows:

Resource Usage (RU): The Resource Usage (RU) is computed as the percentage of HW resources available in the FPGA that have been used to synthesize the chip.

Power Overhead (PO): The Power Overhead (PO) is computed as the overhead due to the addition of HUSTLE on the power consumption of the synthesized chip. We define P as the power consumption of the chip without HUSTLE, and P_h as the Power consumption of the chip implementation integrating HUSTLE; thus, $PO = 1 - (P / P_h)$.

A. HW RESOURCE USAGE

By analyzing the synthesis results, we found that the relative increase in *BoomTile*'s resource usage owing to the integration of the HUSTLE mechanism was relatively low, as shown in Fig. 12. Indeed, the HUSTLE module increases the number of LUTs and FF by less than 1%, whereas DSP and LUTRAM have not increased. The highest increase is of 2.5% in the BRAMs usage, which rises from 3.53% to 6%. This is expected since the HUSTLE module is equipped with an internal memory of 32KB, and the L1 instruction and data cache have both 16KB size.

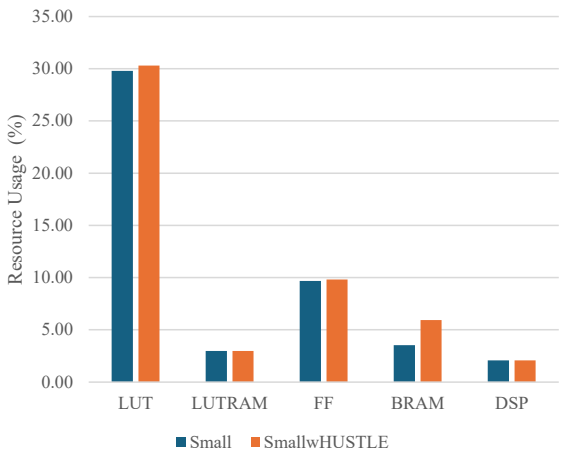

Fig. 12 **Measured FPGA resource usage. The blue columns are used for values measured in a CPU implementation without HUSTLE and the orange column for one with HUSTLE.**

TABLE VIII
CPU LOAD (U) COMPARISON OF HUSTLE WITH A SOLUTION WITH INCREASING IC SIZE ON *SINGLECORE* ARCHITECTURE

SP			10ms			1ms		
FS			850	85	50	850	85	50
SM	Configuration	Baseline						
STD	SingleCoreIC16KB	85.67%	*86.35%*	*90.39%*	*93.44%*	*92.33%*	*OL*	*OL*
STD	SingleCoreIC32KB	85.28%	*85.90%*	*89.95%*	*93.00%*	*91.84%*	*OL*	*OL*
STD	SingleCoreIC64KB	84.95%	*85.66%*	*89.55%*	*92.57%*	*91.46%*	*OL*	*OL*
H_P	SingleCore (IC16KB + HUSTLE 32KB IM)	85.67%	*85.87%*	*86.18%*	*86.36%*	*88.26%*	*90.58%*	*92.13%*

Measurements of CPU load (U) from the test campaign performed on the *SingleCore* architecture to compare HUSTLE with a CPU with larger IC size, OL means, OverLoad, i.e., the CPU load is above 100%.

B. POWER CONSUMPTION

The results of the overhead of HUSTLE on the CPU power consumption are shown in Fig. 13. It can be noted that the overhead on power consumption incurred due to the addition of HUSTLE is around 2% when it is OFF or IDLE. Indeed, the two corresponding lines in the chart are overlapped, from a deeper investigation we found that the most of the power consumption in the these two states is accountable to the IM, since no clock gating techniques have been applied, it is always turned on, and consume some power even when the HUSTLE module is turned OFF. Instead, when the module is in the OPERATIONAL state, the overhead increases to around 12%. As it is shown, increasing the Toggle Rate of the BoomTile input signals does not seems to have significant impact on the power consumption neither on the PO. Note that, the portion of time in which HUSTLE is in the OPERATIONAL state depends on the Safety Period of the STL, and on the time needed to execute it. Hence the overall increase in power consumption it is likely to be lower than the one measured in the OPERATIONAL state.

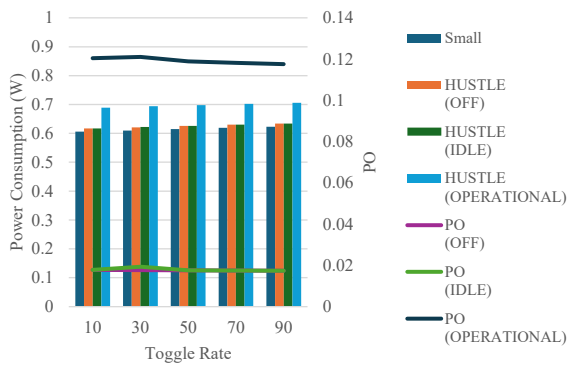

Fig. 13 **Results for Power consumption and PO. The left axis refers to the column chart whereas the right axis refers to the superimposed line chart.**

VII. RELATED WORKS

Permanent HW fault protection techniques for modern CPUs can be resumed into three main categories: Built-in Self-Tests (BIST) [11], Lock-Step[14], and STLs [16]. The high impact on area and power consumption of the former two, and their higher design effort, have led to the spreading of STLs. This widespread use of STLs [16], [23], calls for provisions to diminish the overhead induced by their execution. A common approach is to reduce the test execution time by exploiting test compaction procedures [36]. However, this does not reduce the overhead caused by the interleaving of payload SW, and STL. We found only one study [37] that attempted to achieve this using scheduling mechanisms. The solution proposed in [37] assumes that spare CPUs are available and leverages the OS to perform task migration, while one CPU executes STL test routines. We did not find any work that exploited idle times in the CPU by executing fragments of STL code. Using dedicated HW support to accelerate software operations is a common approach [38]. On this basis, however, we found only two works [39], [40] that adopted dedicated HW support for self-testing by storing STL instructions in dedicated memory. In these studies, the authors focused on implementing hardware support that can store test codes and data. In both studies, the tested CPU transitioned to a test mode that made it unavailable to the system for the entire duration of the tests. In [39] the STL was encoded with specific test generation procedures, and the test mode was entered by means of an ISR. In [40] the test mode required a core to implement checkpointing. Differently from [39] and [40] we did not limit to providing a HW support with dedicated memory, but also the logic to implement specific scheduling strategies (periodic and cache miss-driven), without the need for a dedicated test mode, nor requiring specific HW features to be available, and without imposing constraints on the STL implementation. In TABLE IX a comparison of HUSTLE with state-of-the-art techniques is reported.

VIII. CONCLUSIONS

In this work, we presented HUSTLE, an HW module that allows efficient execution of STL code. We explained and detailed the architecture and behavior of HUSTLE and made experiments to assess to which extent HUSTLE can be used to execute STL instructions without increasing the response

TABLE IX
COMPARISON OF HUSTLE WITH OTHER STATE OF THE ART TECHNIQUES

	HUSTLE	STL	Lock-step	BIST	MHIST [39]	CASP [40]
Area overhead	Low	No	2x	Low-Medium	Low[c]	High
Power overhead	Low	Very Low	Medium	Very High	NA	NA
Performance Overhead	None-Low[a]	Low-Medium[b]	None	High	Low[d]	High[d]

[a] Always lower than standard STL.
[b] Depending on the STL scheduling frequency
[c] Only memory to store a single test, compression is used to store test code and data
[d] Requires detaching the CPU from the system during testing

time of tasks in payload SW. We also evaluated its benefits in terms of CPU load reduction in three different chip architectures using the different STL execution modes provided by HUSTLE.
From the experiments performed, we observed that the use of HUSTLE to execute STL always reduced the CPU Load. The benefits provided by the proposed solution increase when the scheduling frequency of the STL and average cache miss resolution time increase.
The experiments also confirmed that, in most cases, the reduction in CPU utilization can be further improved by using the proposed cache miss-driven interrupt generation mechanism to execute STL instruction, instead of periodic STL scheduling. The benefits of this STL execution mode are strongly dependent on the choice of cache misses and their resolution time. Finally, we considered the impact on the device area and power consumption, showing that HUSTLE has a low impact on the area (most of which is due to the IM for storing the STL Test instructions), and a very limited increase in power consumption, making this solution applicable to systems with a limited budget for power and area.

HUSTLE represents an improvement in the state of the art because it reduces the overhead caused by the interleaving of STL code and payload SW and provides a way to execute STL SW without increasing the task response time. This is achieved with no constraints on the implementation of the STL fragments and with minimal modification to the CPU design.

The proposed solution is flexible, since it is not tied to any specific ISA or CPU architecture. We exploited resources that are generally available in most of the modern CPU cores, hence, we argue that HUSTLE is applicable to many others CPU architectures, which can be used in a large variety of Safety Critical Systems. Moreover, since the STL size is tightly coupled to the amount of logic to stimulate, the size of the IM can be adjusted to fit the needs of the target CPU, allowing our solution to be scaled according to the system complexity.

Additionally, even if HUSTLE has been tailored for STLs, we argue that this kind of mechanism can be applicable to any other non-functional code, e.g., interrupt handlers, pieces of drivers and snippets of hypervisor and trusted firmware code. In general, this mechanism is designed to reduce the overhead caused by the interleaving of functional and non-functional code, and to reduce the inactive processor time. This is done by transforming a periodic task into an aperiodic one and breaking it down into many smaller pieces. Indeed, the choice of applying this execution method to STLs has two main reasons: The first one is that they are an enabling technology used in many safety-critical applications, the second one is that they are conceived to be executed periodically within a well-defined time interval.
Moreover, some features exist that, if available in the CPU design, can enable further improvements of HUSTLE performance benefits. Among these features, we include fast interrupts, to reduce the time for the context switch, and dedicated interrupt channels, as in vectored interrupt-capable CPUs, to allow the use of a separate, optimized ISR for the interrupt requests issued by HUSTLE. Investigating the details of the application of HUSTLE in such scenarios is one of our future directions.

Another direction for improvement is to predict the cache miss resolution time to schedule the execution of STL fragments that can perfectly fit the available resolution time. We conjecture that, if the selection of those cache misses in which to generate interrupts and thus execute STL fragments is carefully performed, it may be possible to execute the entire STL without any increase in task response time.

IX. ACKNOWLEDGEMENTS

We would like to thank Luca Maruccio for the invaluable contributions and support provided during the early stages of the design and prototyping activities.

REFERENCES

[1] F. Rehm et al., "The Road towards Predictable Automotive High - Performance Platforms," 2021 Design, Automation & Test in Europe Conference & Exhibition (DATE), Grenoble, France, 2021, pp. 1915-1924, doi: 10.23919/DATE51398.2021.9473996.
[2] R. Singh and S. S. Gill, "Edge AI: A survey," Internet of Things and Cyber-physical Systems, vol. 3, pp. 71–92, Jan. 2023, doi: 10.1016/j.iotcps.2023.02.004.
[3] A. Lavin et al., "Technology readiness levels for machine learning systems," Nature Communications, vol. 13, no. 1, Oct. 2022, doi: 10.1038/s41467-022-33128-9.
[4] A. Avizienis, J. -c. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," IEEE Transactions on Dependable and Secure Computing, vol. 1, no. 1, pp. 11–33, Jan. 2004, doi: 10.1109/tdsc.2004.2.
[5] "Road vehicles - Functional Safety", ISO 26262:2018 2nd edition, 2018
[6] "Railway applications - The specification and demonstration of Reliability, Availability, Maintainability and Safety (RAMS)", CENELEC EN:50129, 2018
[7] "Electronic Functional Safety Package", IEC EN 61508, 2010
[8] S. Saidi, S. Steinhorst, A. Hamann, D. Ziegenbein and M. Wolf, "Special Session: Future Automotive Systems Design: Research Challenges and Opportunities," 2018 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), Turin, Italy, 2018, pp. 1-7, doi: 10.1109/CODESISSS.2018.8525873
[9] D. P. Siewiorek and R. S. Swarz, "Reliable Computer System Design and Evaluation, Third Edition (3rd ed.)", A K Peters/CRC Press, 1998, doi: 10.1201/9781439863961
[10] B. Littlewood and L. Strigini, "Software Reliability and Dependability: A Roadmap," in Proceedings of the 22nd International Conference on Software Engineering (ICSE2000), Limerick, June 2000, pp. 177-188.
[11] E. J. McCluskey, "Built-In Self-Test Techniques" in IEEE Design & Test of Computers, vol. 2, no. 2, pp. 21-28, April 1985, doi: 10.1109/MDT.1985.294856.
[12] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan and D. I. August, "SWIFT: software implemented fault tolerance" International

Symposium on Code Generation and Optimization, San Jose, CA, USA, 2005, pp. 243-254, doi: 10.1109/CGO.2005.34.

[13] J. H. Wensley et al., "SIFT: Design and analysis of a fault-tolerant computer for aircraft control," in Proceedings of the IEEE, vol. 66, no. 10, pp. 1240-1255, Oct. 1978, doi: 10.1109/PROC.1978.11114.

[14] X. Iturbe, B. Venu, E. Ozer, J.-L. Poupat, G. Gimenez, and H.-U. Zurek, "The Arm Triple Core Lock-Step (TCLS) Processor," ACM Trans. Comput. Syst., vol. 36, no. 3, pp. 1–30, 2018. DOI: 10.1145/3323917.

[15] D. Kuvaiskii, R. Faqeh, P. Bhatotia, P. Felber and C. Fetzer, "HAFT: hardware-assisted fault tolerance", Proc. 11th Eur. Conf. Comput. Syst., pp. 1-17, Apr. 2016, doi: 10.1145/2901318.2901339

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

[17] N. Hage, R. Gulve, M. Fujita and V. Singh, "On Testing of Superscalar Processors in Functional Mode for Delay Faults," 2017 30th International Conference on VLSI Design and 2017 16th International Conference on Embedded Systems (VLSID), Hyderabad, India, 2017, pp. 397-402, doi: 10.1109/VLSID.2017.58.

[18] "Software Test Library NXP". Accessed 27/06/2024. [online] Available: https://www.nxp.com/design/design-center/software/functional-safety-software/structural-core-self-test-scst-library:SCST

[19] "Software Test Library STMicroelectronics". Accessed 27/06/2024. [online] Available: https://www.st.com/en/embedded-software/x-cube-classb.html

[20] "Software Test Library Renesas". Accessed 27/06/2024. [online] Available: https://www.renesas.com/en-eu/products/synergy/software/add-ons.html#read.

[21] "Software Test Library Microchip". Accessed 27/06/2024. [online] Available: https://www.microchip.com/en-us/products/microcontrollers-and-microprocessors/32-bit-mcus/32-bit-functional-safety/industrial-safety-self-test-library

[22] "Software Test Library ARM". Accessed 27/06/2024. [online] Available: https://www.arm.com/products/development-tools/embedded-and-software/software-test-libraries

[23] F. Pratas, T. Dedes, A. Webber, M. Bemanian and I. Yarom, "Measuring the effectiveness of ISO26262 compliant self test library," 2018 19th International Symposium on Quality Electronic Design (ISQED), Santa Clara, CA, USA, 2018, pp. 156-161, doi: 10.1109/ISQED.2018.8357281.

[24] Y. K. Malaiya, Naixin Li, J. Bieman, R. Karcich and B. Skibbe, "The relationship between test coverage and reliability," Proceedings of 1994 IEEE International Symposium on Software Reliability Engineering, Monterey, CA, USA, 1994, pp. 186-195, doi: 10.1109/ISSRE.1994.341373.

[25] E. F. Weglarz, K. K. Saluja and T. M. Mak, "Testing of hard faults in simultaneous multi-threaded processors" Proceedings. 10th IEEE International On-Line Testing Symposium, Funchal, Portugal, 2004, pp. 95-100, doi: 10.1109/OLT.2004.1319665.

[26] N. Ferrante, F. Terrosi, L. Maruccio, F. Rossi, L. Fanucci, and A. Bondavalli, "HUSTLE: A Hardware Unit for Self-test-Libraries Efficient Execution", in Applications in Electronics Pervading Industry, Environment and Society, 2024, pp. 392–398. doi: 10.1007/978-3-031-48121-5_56

[27] B. Sprunt, L. Sha, and J. Lehoczky, 'Aperiodic task scheduling for Hard-Real-Time systems', Real-Time Systems, vol. 1, no. 1, pp. 27–60, Jun. 1989.

[28] T.-H. Lin and W. Tarng, "Scheduling periodic and aperiodic tasks in hard real-time computing systems", in Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, San Diego, California, USA, 1991, pp. 31–38.

[29] M. Spuri and G. Buttazzo, "Scheduling aperiodic tasks in dynamic priority systems", Real-Time Systems, vol. 10, no. 2, pp. 179–210, Mar. 1996.

[30] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio et al., "The Rocket Chip Generator", EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17, 4, 6-2. Apr 2016.

[31] "Chipyard's documentation". Accessed 27/6/2024. [Online] https://chipyard.readthedocs.io/en/stable/

[32] A. Waterman et al., "The RISC-V instruction set manual", Volume I: User-Level ISA', 2014.

[33] "BOOM Core". Accessed 27/06/2024. [Online] Available: https://boom-core.org/

[34] C. Celio, D. A. Patterson, and K. Asanović, "The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor", Jun. 2015.

[35] "RISCV-BOOM's documentation". Accessed 27/06/2024. [Online] Available: https://docs.boom-core.org/en/latest/

[36] M. Gaudesi, I. Pomeranz, M. S. Reorda and G. Squillero, "New Techniques to Reduce the Execution Time of Functional Test Programs," in IEEE Transactions on Computers, vol. 66, no. 7, pp. 1268-1273, 1 July 2017, doi: 10.1109/TC.2016.2643663

[37] Y. Li, O. Mutlu, and S. Mitra, 'Operating system scheduling for efficient online self-test in robust systems', in Proceedings of the 2009 International Conference on Computer-Aided Design, San Jose, California, 2009, pp. 201–208.

[38] W. J. Dally, Y. Turakhia, and S. Han, 'Domain-specific hardware accelerators', Commun. ACM, vol. 63, no. 7, pp. 48–57, Jun. 2020.

[39] P. Bernardi, L. M. Ciganda, E. Sanchez and M. S. Reorda, "MIHST: A Hardware Technique for Embedded Microprocessor Functional On-Line Self-Test," in IEEE Transactions on Computers, vol. 63, no. 11, pp. 2760-2771, Nov. 2014, doi: 10.1109/TC.2013.165.

[40] Y. Li, S. Makar and S. Mitra, "CASP: Concurrent Autonomous Chip Self-Test Using Stored Test Patterns," 2008 Design, Automation and Test in Europe, Munich, Germany, 2008, pp. 885-890, doi: 10.1145/1403375.1403590.

This article has been accepted for publication in IEEE Access. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2024.3434718

NICOLA FERRANTE received the B.S. degree in computer engineering from University of Pisa, in 2019, and, in 2021, the M.S. degree in computer engineering from University of Pisa.
He is currently pursuing the Ph.D. degree in Information Engineering at University of Pisa. From 2021, he works as researcher in Resiltech SRL.
His research interests include the development of efficient SW-based fault protection mechanisms, fault models for aging-related faults in critical systems, and fault detection systems that exploit artificial intelligence algorithms.
Mr. Ferrante is a member of ISO/TC 22/SC 32/WG 8 and WG13 and has participated in the development of the ISO PAS 8926 – Functional Safety - Reuse of Pre-existing Software.

LUCA FANUCCI Luca Fanucci (Fellow, IEEE) received the Laurea and Ph.D. degrees in electronic engineering from the University of Pisa, in 1992 and 1996, respectively. From 1992 to 1996, he was with the European Space Agency—ESTEC, Noordwijk, The Netherlands, as a Research Fellow. From 1996 to 2004, he was a Senior Researcher with the Italian National Research Council in Pisa. He is currently a Professor in microelectronics with the University of Pisa. He is the coauthor of more than 500 journal articles and conference papers and a co-inventor of more than 40 patents. His research interests include several aspects of design technologies for integrated circuits and electronic systems, with particular emphasis on system-level design, hardware/software co-design, and sensor conditioning and data fusion. His main applications areas are in the field of wireless communications, low-power multimedia, automotive, healthcare, ambient assisted living, and technical aids for independent living. He is a member of the editorial board of Technology and Disability (IOS Press). He is a fellow of DATE. He served in several technical programme committees for international conferences. He was the Program Chair of DSD 2008 and DATE 2014 and the General Chair of DATE 2016 and HIPEAC 2020.(Based on document published on 30 March 2023).

FRANCESCO ROSSI took his Master degree in Electronic Engineering with 110/110 cum Laude at University of Pisa (Pisa, Italy) in 2002 and his PhD degree in Information Engineering at Department of Information Engineering of University of Pisa (Pisa, Italy) in 2007 with curriculum "Micro and nanoelectronic technologies, devices and systems". In the years 2004-2007 he published 16 papers in international journals and conference proceedings mainly focusing on algorithms and VLSI architectures for telecom applications. From 2007 to 2010 he worked in Renesas Electronics as Senior LSI designer for Automotive microcontrollers for safety-relevant applications. Since 2011 he is the Automotive Safety Solution Manager in Resiltech, and in these years, he acted as safety manager in projects for Tier1 companies and supported OEM, Tier2 and component provider in implementing a number of project compliant with ISO26262. As safety expert he is joining ISO WG8 and WG13 activities.

FRANCESCO TERROSI received his master's degree in computer science (curriculum "Resilient and secure cyber-physical systems") from the University of Florence, Italy, in 2020. He is currently a PhD student at the same university. His research activities are mostly centered on safety-critical systems, with a focus on their properties such as safety, fault tolerance and, more in general, dependability of such systems and their hardware components. His other research interests include Machine Learning, from the theoretical aspects to their application in safety-critical systems, and hence all the aspects related to safety assurance of such components.

ANDREA BONDAVALLI Andrea Bondavalli (Senior Member, IEEE) is currently a Full Professor in computer science with the University of Florence. Previously, he has been a Researcher and a Senior Researcher of the Italian National Research Council, working at the CNUCE Institute in Pisa. In particular, he has been involved in safety, security, fault tolerance, evaluation of attributes, such as reliability, availability, and performability. His scientific activities have originated more than 220 papers appeared in international journals and conferences. He supports as an Expert of the European Commission in the selection and evaluation of project proposals and regularly consults companies in the application field. His research interest includes the dependability and resilience of critical systems and infrastructures. He participates to (and has been chairing) the program committee in several international conferences, such as IEEE FTCS, IEEE SRDS, EDCC, IEEE HASE, IEEE ISORC, IEEE ISADS, IEEE DSN, and SAFECOMP.