







HUSTLE: A Hardware Unit for Self-test-Libraries Efficient Execution

Nicola Ferrante^{1,2} , Francesco Terrosi³ , Luca Maruccio¹, Francesco Rossi¹,
Luca Fanucci² , and Andrea Bondavalli³ 

¹ Resiltech, 56025 Pontedera, PI, Italy

nicola.ferrante@resiltech.com

² University of Pisa, 56126 Pisa, PI, Italy

³ University of Florence, 50121 Firenze, FI, Italy

Abstract. Online testing of computer systems is crucial in contexts such as the safety-critical domain, where the software is usually made of functional code, which is the code implementing the application-specific functionalities, and non-functional code, which implements auxiliary functionalities, e.g., test routines. By periodically running a test routine it is possible to satisfy the high dependability requirements mandated by regulators, and defined in safety standards such as ISO26262, IEC61508, and CENELEC EN 5012X. Self-Test Libraries (STLs) are a form of software-based self-test, widely used in safety-related applications. The main drawback of this safety mechanism is the overhead imposed on the execution of the functional code, and reducing this overhead is a well-known challenge in research. We propose here HUSTLE, a Hardware Unit for STL Efficient execution, which can be integrated into the chip design with no modification to the CPU's internal logic. We also propose a scheduling mechanism that allows HUSTLE to efficiently execute self-tests, by exploiting the CPU's idle time. This is achieved by storing test code in a separate memory and sending instructions to the CPU, bypassing the Instruction Cache, thus allowing to reduce the overall execution time and the cache interference of STL, while CPU utilization increases.

Keywords: Software-based self-test · Safety-critical systems · Embedded systems testing

1 Introduction

In safety-critical systems, protecting the CPU from hardware faults in the field is a fundamental requirement [1, 2]. Many protection techniques were proposed, based on both hardware (HW) and software (SW) mechanisms, each of them providing different levels of protection. HW-based techniques are faster but require modifications to the original design, whereas SW-based techniques have little to no impact on the device area, but significantly reduce performance [3, 4]. To reduce the impact of safety mechanisms on performance, we propose an approach that exploits the idle time of the CPU to execute test routines, i.e., Self-Test Libraries (STL). Well-established Functional Safety standards, such as ISO 26262, require high coverage of random hardware faults, for which

STLs have been identified as an effective safety mechanism, providing high coverage without any impact on the device's design [5]. However, to achieve the target protection level STLs need to stimulate as much as possible the internal logic of the component, imposing a significant overhead on the nominal execution [6]. We present here HUSTLE, a hardware mechanism to improve the execution efficiency of STLs. HUSTLE provides STL instructions to the core without accessing the Instruction Cache (IC), reducing the overall response time and the IC pollution (cache misses for STL instructions are eliminated), since the cache will keep in memory only instructions relative to the functional workload. To demonstrate the possibilities offered by HUSTLE we also implemented an efficient event-triggered scheduling mechanism by exploiting architectural signals to detect CPU's idle time, and use this time to execute STL instructions. HUSTLE allows to reduce the overhead on the execution time and the interference with the cache, increasing core utilization.

2 Related Works

The use of instruction-based self-test, i.e., STL, is a widely used technique [7], and improving their efficiency is an active research field. Most designers of such test libraries try to leverage target architecture's resources to achieve a higher protection degree while meeting constraints on the imposed overhead [8, 9], while others try to develop finer algorithms that are both general and efficient. E.g., in [10] the authors merge different SW techniques combining random program generation and signature-based self-checking; in [11] the authors focus on improving the automatic generation of test programs, while [12] proposes a technique that relies solely on available processor resources. The scheduling of such tests is crucial to their efficiency and effectiveness [9, 13]. We found no study about scheduling mechanisms that exploit idle times in the CPU by executing fragments of STL code. Using dedicated HW support to speed up software operations is a common approach [14], however, we found only one work that adopts this approach towards self-testing, by storing STL instructions in a dedicated memory [15]. In this work, the authors focus on implementing hardware support that can store test code and data. The STL is implemented as an Interrupt Service Routine, but no specific scheduling strategy is proposed.

3 HUSTLE Overview

3.1 Mechanism Description

HUSTLE is placed between the CPU and the IC, to provide STL instructions to the Core as fast as possible. This is achieved as follows: during the execution of functional code, HUSTLE, is in IDLE state, and it just forwards requests received from the CPU to the IC, and the subsequent responses from the IC to the CPU. When the CPU starts executing the STL it requests addresses in the address range of HUSTLE's ROM, HUSTLE transitions into the ACTIVE state. In this state, it blocks requests from the CPU to the IC and responds to the CPU with the requested instructions in place of the IC. Internally, HUSTLE has a Read Only Memory (ROM) containing the STL payload, i.e., the set of

STL instructions that must be executed. A high-level schema of HUSTLE’s interface is depicted in Fig. 1a, while the transition diagram is shown in Fig. 1b. This addition to the design allows to: (i) reduce STL instruction latency, as STL instructions are not fetched from the IC or lower levels of cache, (ii) reduce IC pollution, since a request to an address relative to an STL instruction will be handled by a separate memory, and (iii) increase overall core’s resource usage, as result of (i) and (ii).

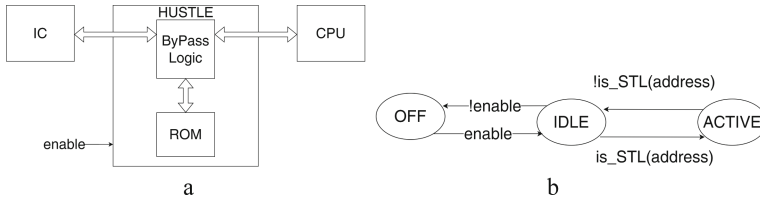


Fig. 1. **a** HUSTLE block diagram, **b** HUSTLE transition diagram.

3.2 HUSTLE Scheduling Strategy

HUSTLE’s scheduling strategy is based on monitoring architectural signals to detect events that may cause instruction starvation and leave the core idle. One of the most frequent causes of instruction starvation are IC misses [16]. When a miss in the IC happens, the core must wait for its resolution to execute the next instruction. HUSTLE can exploit this time to execute STL instructions stored in its ROM.

The proposed scheduling strategy triggers a control-flow redirection sending a hardware interrupt as soon as a cache miss is detected. In our prototype, the IC miss signal is routed to HUSTLE and connected to the interrupt controller, using the STL as Interrupt Service Routine (ISR). Note that, without ad-hoc HW support it wouldn’t be possible to exploit idle times caused by cache misses, because the control flow redirection to handle the interrupt could cause additional cache misses itself, thus invalidating all the benefits. In Fig. 2 we illustrate a simple scheduling example showing how idle times could be leveraged by HUSTLE. It is important to note that the STL code shall take into account proper mechanisms to avoid interference with other running applications. For example, context-switching techniques, granularity of sequences of consecutive STL tests to execute, and their priority. In general, all the aspects that can affect the impact of this scheduling strategy on the execution time of other applications should be analyzed, based on the requirements of the target system. The analysis of these aspects is not reported in this work for brevity.

4 Experimental Activity

To evaluate HUSTLE, a prototype has been developed using the Chipyard [17] framework. We chose as target CPU architecture the RISC-V [18] Berkley Out-of-Order Machine (BOOM) Core [19]. To evaluate the impact of the developed solution on different hardware configurations, we selected two configurations of the BOOM core:

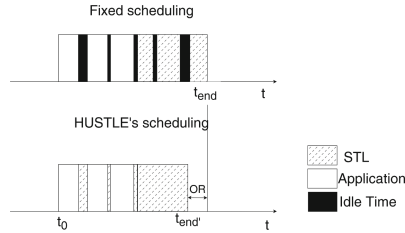


Fig. 2. Visual representation of HUSTLE's scheduling strategy. OR stands for overhead reduction.

SmallBoom, a single pipeline core, and the MediumBoom, a two-wide pipeline core. The Verilog RTL description of the design is generated using Chipyard's toolchain, which was then compiled and simulated using Synopsys VCS.

HUSTLE is placed between BOOM's Frontend (Containing the IC), and the Core. A Control Status Register (CSR) was created to allow enabling HUSTLE via SW. HUSTLE's ROM is initialized with STL instructions, which are mapped at a specific address that is configured at runtime as the ISR address for HUSTLE's interrupt.

The SW used for evaluation is made of three components: the workload, the STL, and a scheduler function. The workload is made of integer and floating-point operations e.g., addition, multiplication, and bit shifts, and it is large enough to fill the IC, to guarantee IC misses to happen during its whole execution. The scheduler function consists of a for-loop with a variable number of iterations. Inside the for-loop, the workload is scheduled once for each iteration. The STL is composed of 10 signature-based arithmetic and logic tests developed in assembly, and it includes a context switch routine to preserve application(s) context. We developed three variations of the code which we call: test_1, test_2, and test_3. In test_1 STL instructions are scheduled after each workload execution. Here we intend to model the worst-case for STL execution, where the whole IC is filled with functional-code instructions. In test_2, the STL is scheduled periodically, with higher priority than the workload. This test is used to evaluate the effect of HUSTLE in a common-case scenario, in which some STL instructions may be already in the IC when the STL starts its execution. Finally, in test_3 we evaluate HUSTLE's scheduling strategy, against a fixed periodic scheduling strategy. Each software is tested on the BOOM Core as is, and the BOOM Core with HUSTLE. Hereafter we define a set of metrics to evaluate HUSTLE. Values measured on HW configuration with HUSTLE are reported with the h subscript.

Execution overhead of STL (OR): To evaluate the reduction of the overhead caused by the STL, we define C as the number of additional clock cycles required to execute the STL w.r.t. the baseline. Overhead reduction is computed as $OR = 1 - (C_h/C)$.

CPU usage during STL (ΔIPC): To evaluate HUSTLE's effect on the CPU resource usage we measure the Instructions Per Cycle (IPC) on the overhead imposed by the STL. We define N as the number of executed STL instructions, thus $IPC = N/C$. Then we measure the difference on HW configurations with and without HUSTLE as $\Delta IPC = IPC_h - IPC$.

Cache Interference of STL (IR): To measure the cache interference of the STL we define M as the number of additional cache misses caused by STL instructions w.r.t the baseline. We compute the interference reduction as $IR = I - (M_h/M)$.

5 Results

In this section, we discuss the results obtained from the experimental campaign. Data are reported in Table 1.

Table 1. Metrics for *test_1* and *test_2*.

SW	HW	OR	Δ IPC	IR
<i>test_1</i>	Small	0.355	0.278	1.026
	Medium	0.496	0.477	0.907
<i>test_2</i>	Small	0.032	0.027	0.953
	Medium	0.058	0.069	1.035

We see that HUSTLE provides an OR up to 40% on the worst-case scenario. Whereas in the common case, i.e., *test_2*, the benefits are reduced. This may be because STL instructions are still present in the cache, hence the use of HUSTLE has not a considerable impact as in *test_1*. We note also that the improvement provided by HUSTLE is higher with the MediumBoom configuration than with the SmallBoom in both tests. The increase in resource usage of the CPU (column Δ IPC) is much higher in *test_1*; this is expected since the idle time due to cache misses is reduced and instructions are served directly by HUSTLE. The MediumBoom approximately doubles HUSTLE’s benefits w.r.t. SmallBoom configuration. The results on cache interference (column IR) present a different trend than the others. As can be noted, the SmallBoom has a higher benefit in this case: in *test_1* the number of cache misses is less than the baseline ($IR > 1$), whereas in the MediumBoom is always lower than 0.91. We argue that this may be due to branch predictor and speculative execution.

Having confirmed the benefits of HUSTLE, we now compare two different scheduling strategies: *periodic* at a fixed time interval and *event-driven* in correspondence of cache misses. Values are reported in Table 2. Looking at the rows “HUSTLE” and “Periodic”, results show that the proposed scheduling strategy outperforms the fixed scheduling strategy in both Small and Medium configurations. We can see that HUSTLE’s scheduling strategy almost doubles Δ IPC and OR for the Small configuration, while more than doubling these values for the Medium configuration. The Medium configuration shows the larger benefits, approximately doubling the increase in IPC w.r.t. a fixed scheduling strategy, and the same applies to the OR.

Table 2. *test_3* results, the baseline execution considered is *test_2*.

HW	Scheduling	Δ IPC	OR
Small	HUSTLE	0.049	0.057
	Periodic	0.027	0.032
Medium	HUSTLE	0.133	0.107
	Periodic	0.069	0.058

6 Conclusions

In this work, we presented HUSTLE, a HW module that allows efficient execution of STL code. We showed how with this module it is possible to reduce the overhead and cache interference of STL execution while increasing the CPU utilization. We also provide an implementation of an efficient event-triggered STL scheduling mechanism to show the benefits provided by HUSTLE. In our experiments, we demonstrated that this mechanism increases CPU utilization (IPC) while reducing the overhead on execution time. Moreover, we found that by using HUSTLE it's possible to reduce the interference between functional and non-functional code, in particular the allocation of cache lines for non-functional code, and in some cases reducing also the cache misses in functional code.

References

1. Peleska J, Siegel M (1996) Test automation of safety-critical reactive systems
2. Parnas DL et al (1990) Evaluation of safety-critical software. ACM
3. Osinski L, Langer T, Mottok J (2017) A survey of fault tolerance approaches on different architecture levels. ARCS
4. Abid A, Khan MT, Iqbal J (2021) A review on fault detection and diagnosis techniques: basics and beyond. Artif Intell Rev 54
5. Pratas F et al (2018) Measuring the effectiveness of ISO26262 compliant self-test library. In: 2018 ISQED. IEEE
6. Malaiya YK et al. The relationship between test coverage and reliability. In: Proceedings of 1994 IEEE international symposium on software reliability engineering
7. Psarakis M et al (2010) Microprocessor software-based self-testing. IEEE Des Test Comput 27(3):4–19
8. Bernardi P et al (2015) Development flow for online core self-test of automotive microcontrollers. IEEE Trans Comput 65(3):744–754
9. Paschalis A, Gizopoulos D (2004) Effective software-based self-test strategies for on-line periodic testing of embedded processors. IEEE Trans Comput Aided Des Integr Circuits Syst 24(1):88–99
10. Kranitis N et al (2008) Hybrid-SBST methodology for efficient testing of processor cores. IEEE Des Test Comput 25(1):64–75
11. Riefert A et al (2016) A flexible framework for the automatic generation of SBST programs. IEEE Trans on Very Large Scale Integr (VLSI) Syst 24(10)
12. Hatzimihail M et al (2007) A methodology for detecting performance faults in microprocessors via performance monitoring hardware. In: 2007 IEEE international test conference

13. Li Y et al (2009) Operating system scheduling for efficient online self-test in robust systems. In: International conference on computer-aided design
14. Dally WJ, Turakhia Y, Han S (2020) Domain-specific hardware accelerators. *Commun ACM* 63(7):48–57
15. Bernardi P et al (2013) MIHST: a hardware technique for embedded microprocessor functional on-line self-test. *IEEE Trans Comput* 63(11):2760–2771
16. Kanev S et al (2015) Profiling a warehouse-scale computer. In: Proceedings of the 42nd annual international symposium on computer architecture
17. Chipyard. <https://chipyard.readthedocs.io/en/stable/>. Accessed 28 Mar 2023
18. RISC-V ISA. <https://riscv.org/>. Accessed 28 Mar 2023
19. BOOM Core. <https://boom-core.org/>. Accessed 28 Mar 2023