POLITECNICO DI TORINO
Repository ISTITUZIONALE

A Flexible Framework for Vector Accelerators In-field Testing

Original
A Flexible Framework for Vector Accelerators In-field Testing / Vilar De Farias, G., Rodriguez Condia, J.E., Sonza Reorda, M.. - (2026). (44th IEEE VLSI Test Symposium Napa, CA (USA) 27-29 April 2026) [10.1109/VTS69484.2026.11563325].

Availability:
This version is available at: 11583/3007196 since: 2026-10-23T11:54:09Z

Publisher:
IEEE

Published
DOI:10.1109/VTS69484.2026.11563325

Terms of use:

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

Publisher copyright
IEEE postprint/Author's Accepted Manuscript

©2026 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collecting works, for resale or lists, or reuse of any copyrighted component of this work in other works.

(Article begins on next page)

20 November 2026

A Flexible Framework for Vector Accelerators In-field Testing

Gustavo Vilar de Farias*, Josie E. Rodriguez Condia*, Matteo Sonza Reorda*
*Politecnico di Torino - Department of Control and Computer Engineering (DAUIN)
{gustavo.vilarde, josie.rodriguez, matteo.sonzareorda}@polito.it

***Abstract*—[1] Advances in technology nodes, activity stress caused by data-intensive workloads, and harsh operating conditions increase the occurrence of faults in edge devices used in Artificial Intelligence domains, severely impacting their reliability. Among diverse accelerators, Vector Processing Units (VPUs) offer a versatile and flexible architecture for accelerating workload execution by exploiting data-level parallelism.**

This paper targets fault detection for VPUs and describes a method for generating Software Test Libraries (STLs) to detect permanent faults arising during the in-field operational phase. The method generates an STL for the baseline accelerator and customizes it for any VPU configuration. The STL for the baseline VPU configuration is built by first exploiting application code, identifying suitable data patterns to improve fault detectability, and then adding a few Ad Hoc routines. The resulting test, once the target hardware parameters are known, is customized using the proposed framework to ensure its effectiveness on the final VPU. The time constraints of in-field tests are taken into account by minimizing the test duration.

The experimental results, obtained on a configurable VPU for edge AI applications (Klessydra-T), show that our framework supports the development of effective STLs (up to 96% stuck-at Fault Coverage (FC) for large size configurations). Remarkably, we observed that using a generic STL without accounting for the specific hardware configuration can cause an FC drop of about 25%, highlighting the need for a framework that automatically customizes the test to the VPU's parameters. The framework is particularly attractive for scalable devices where long-term reliability is essential, and tests must be designed for different hardware configurations.

***Index Terms*—Edge AI, Functional testing, Tensor operations, Vector accelerators, Software-Based Self-Test, Reliability**

I. Introduction

Advances in miniaturization and power efficiency of modern semiconductor technologies support continuous innovations in computing devices, allowing the implementation of complex applications, including those using Artificial Intelligence (AI) algorithms. However, conventional processor architectures are often not suitable for such heavy computational workloads. Thus, modern devices boost performance through the integration of specialized hardware (HW) accelerators, such as those based on *Single-Instruction Multiple-Data/Thread* (SIMD/SIMT) paradigms or on systolic array architectures [1]. In these scenarios, vector accelerators (also known as *Vector Processing Units*, or VPUs) represent versatile and power-efficient architectures. Due to their features, they have been successfully used as coprocessors and in-chip accelerators in several modern generations of processors, e.g., introducing scalable and multi-threading vector extensions [2] [3], [4]. More in detail, VPUs overcome limitations of purely SIMD architectures by improving performance in inherently data-parallel applications, like AI inference, digital signal processing, and scientific computing [5], [6].

Modern AI applications are inherently characterized by high data intensity and repetitive tensor/linear algebra operations, which may over-stress the internal structures of an accelerator and increase the probability of hardware faults during the in-field operation of a system. Previous works highlighted the functional fault criticality and detection for AI accelerators [7], [8]. In addition, several studies [9], [10] highlight that advanced technology nodes (below 10nm) are more highly susceptible to temporal and environmental variations, such as stressful operating conditions, temperature fluctuations, and premature aging. Together, these phenomena may induce permanent hardware faults during in-field operation, leading to performance degradation and *Silent Data Corruption* (SDC) effects [11], [12], producing erroneous results that may become especially critical in safety environments.

To address this in-field fault-detection challenge, several alternatives have emerged (e.g., design for testability and software-based self-test). Unfortunately, design for testability (DfT) solutions for testing (e.g., scan, or Logic BIST) are sometimes commonly unavailable (e.g., because too expensive) or unsuitable to detect these faults in the field (highly intrusive). Hence, their usage for in-field tests may be critical or even unfeasible, especially for real-time, mission-critical AI applications.

Instead, software-based self-test (SBST) solutions enable continuous system operation while identifying faults through periodic testing. These solutions support in-field test with minimal performance degradation and without any design modification (and thus area overhead). Generally, SBST strategies use specially crafted test programs (known as *Software Test Libraries*, or *STLs*) [13]–[15]: STL execution typically occurs at power-on, or during the application's idle times, thus minimally impacting the system's availability and performance. Other strategies leverage the application code by resorting to cleverly selected input data that can detect hardware faults [16].

[1]This work was partially supported by the TIRAMISU project under the EU Grant#101169378 — HORIZON-MSCA-2023-DN-01, and by the DARE SGA1 project, funded by the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 101202459

979-8-3315-6337-0/26/$31.00 ©2026 IEEE

The flexibility of most accelerator architectures allows the VPU to be used in diverse scenarios and with different configurations to meet the application's needs, either by changing the number of functional units (FUs) to match workload vector sizes or by exploiting application parallelism across multiple VPU instances. However, this hardware flexibility creates a significant challenge for STL generation. In fact, an STL targeting one specific configuration (e.g., single-core VPU) is often ineffective for another (e.g., multi-core VPU), and generating from scratch a suitable STL for a specific VPU scenario may become significantly time-consuming. This underscores the need for solutions that can systematically customize the STL across the wide range of VPU configurations without the high cost of manual redevelopment.

To address this challenge, this work presents a methodology to first generate STLs for the in-field test of permanent faults arising in a baseline vector accelerator configuration. In detail, the test is developed by first exploring input data variation in applications that naturally exercise the accelerator, reusing some available application code. Then, the method adds a few Ad Hoc routines to detect the hard-to-detect faults. Furthermore, the proposed approach leverages multi-threading to execute different test code segments concurrently, thereby increasing fault detection capabilities while minimizing test execution time. Finally, to address the highly configurable VPU challenge, we developed a framework, named **PIPPO** (*Parameterizable In-field vector Processor test Program generatOr*), that automatically adapts the baseline STL to maintain high fault coverage across various hardware configurations.

For the experimental evaluation of the approach, we implemented the proposed technique on *Klessydra-T* [5], a highly flexible VPU specifically designed for edge computing, targeting permanent stuck-at faults (SAFs). For the STL generation, we first used some application code that implements basic tensor/linear algebra operations, such as matrix multiplication and convolution workloads. Moreover, suitable test data and additional Ad Hoc routines have been developed by identifying input data patterns that maximize coverage and then writing suitable code fragments to target the remaining faults. The results indicate that around 94% of permanent faults in the baseline VPU can be detected through our non-intrusive method with zero area overhead. Our method also allows generating an effective STL without requiring an ATPG (Automatic Test Pattern Generation) step, i.e., avoiding the complex translation of ATPG-generated patterns into input data [17]. Our approach starts with RTL abstraction, leveraging faster simulation (up to $4.1\times$ with respect to gate level) to estimate preliminary fault coverage and rapidly explore the effectiveness of different input data patterns. This can be done even before the gate-level VPU model is available. Then, the STL is tuned and evaluated at the gate level (GL) for the baseline hardware version (accelerator with the smallest configuration possible). Finally, we introduce the **PIPPO** framework for automatically customizing the STL for more complex VPU configurations by scaling test patterns and code to the target HW parameters. Using **PIPPO** to customize the STL allows achieving an increase of up to 25% in the FC for some VPU configurations with respect to simply using the baseline STL.

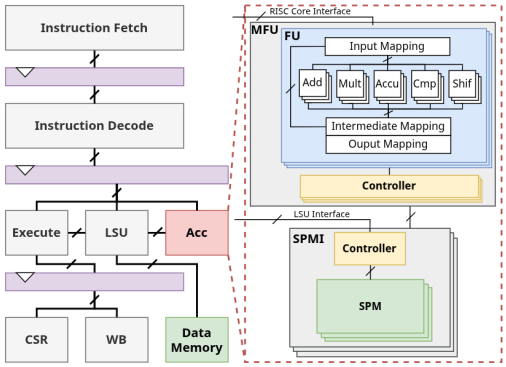


Fig. 1. A general scheme of the Klessydra-T block organization.

The first contribution of this work lies in a methodology for generating STLs for a baseline VPU. A second key contribution is a framework for automatically customizing the STL to a given configuration, enabling the automated generation of effective in-field tests for all hardware configurations.

The document is organized as follows. Section II briefly describes the architecture of VPUs and *Klessydra-T*, and the STL fundamentals. Section III describes the proposed methodology for generating suitable STLs for in-field testing and the framework for customizing the STLs. Section IV details the experimental setup. Section V reports and analyzes the results, and Section VI concludes the work and outlines directions for future work.

II. Background

A. Organization of vector accelerators

Vector accelerators, or VPUs, have been used to improve performance in supercomputers for decades [18]. Now, advances in semiconductor technology enable their use in general-purpose computing [19], resource-constrained edge devices [5], and High-Performance Computing (HPC) systems [20]. In general, VPUs exploit data parallelism through the SIMD paradigm and comprise one or more cores, namely *lanes*, each integrating multiple FUs. Most VPUs are integrated into the CPU's pipeline to increase execution performance through specialized instructions and ISA extensions (e.g., RISC-V V, or Intel AVX). This architectural versatility improves performance across a wide range of applications by enabling configurable data types and the execution of variable-length vector operations. Indeed, the parallelism of VPUs matches the computational demands of AI workloads, which are dominated by matrix multiplications and convolutions, accounting for roughly 90% of operations [21]. VPUs provide a compelling balance of performance, area, and energy efficiency through simple control logic that distributes operations across FUs. Data is first loaded from main memory into the VPU memory, then processed concurrently on the *lanes*, with vector instructions executed iteratively within the FUs.

Klessydra-T [5] is an energy-efficient and open-source VPU for resource-constrained edge devices. This VPU is

a configurable RISC-V core with a custom multi-functional unit (MFU) based on an interleaved multi-threading (IMT) architecture and a custom vector instruction set. The MFU supports both dedicated per-thread implementations and shared multi-thread configurations with a configurable number of FUs and scratchpad memory (SPM), as depicted in Fig. 1. The multi-stage architecture in FUs maximizes the operational frequency [5]. In addition, the multiplier unit comprises four dynamically configurable 16-bit multipliers to support integer or fixed-point formats in 3 multi-precision modes: one 32-bit, two independent 16-bit, or four partitioned 8-bit operations.

The reliability of VPUs has become increasingly relevant due to their extensive use in safety-critical environments. Previous works have proposed fault-tolerant versions of VPUs [22]–[25]; however, these solutions focus on transient faults and are not suitable for permanent faults. Moreover, they typically involve significant overhead in terms of silicon area.

B. Self-Test Libraries

A STL is a set of software routines used to verify hardware functionality during operation by stressing different components and comparing the results with expected values. Any mismatch can trigger an exception or flag to indicate potential hardware faults. Unlike post-manufacturing tests, STLs support in-field test without external equipment [13]. They are widely adopted in safety-critical domains, where functional safety standards, such as ISO 26262, mandate periodic verification, and have recently gained relevance in HPC domains, e.g., through *Software Diagnostic Libraries* [26].

Some STL generation strategies implement specific deterministic algorithms that are known to detect faults in a given unit [27]. Other strategies use pseudo-random approaches to add or modify instructions and data to extend the fault detection features [28]. When targeting embedded systems, some strategies take advantage of the application itself as the base for fault detection [17].

Our approach focuses on the STL development for a baseline VPU by considering first existing applications, and by improving their fault detection capabilities via carefully selected *test data*. Furthermore, to address a few hard-to-detect faults, the method uses some specific Ad Hoc routines. Finally, **PIPPO** customizes the STL to any specific VPU configuration.

III. Proposed Methodology

The proposed methodology targets the in-field test of permanent SAFs in VPU's through the development of STLs customized according to micro-architecture features of a VPU. In particular, our method first takes advantage of existing applications on a baseline VPU to generate an STL that cleverly uses selected input data to increase fault detection capabilities. To increase test development speed, we employ a two-phase abstraction strategy. First, the exploration process is performed at the RT level to accelerate the identification of the most effective test patterns by reducing simulation time. Second, once the fault coverage saturation point is reached, we perform gate-level evaluations. This phase validates the overall

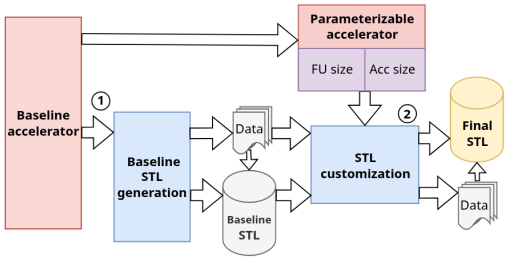


Fig. 2. A general scheme of the proposed methodology.

FC and refines it by adding complementary Ad Hoc routines to the STL to support the test of hard-to-detect faults. Finally, the **PIPPO** framework is used to automatically customize the STL. It adapts the original test patterns and code to match the specific parameters of the target HW configuration, and the results are validated.

Our strategy consists of 2 main steps: *i)* Baseline STL generation, *ii)* STL customization, as depicted in Figure 2 with the key contributions of this work highlighted in blue. The following subsections describe each step.

A. Baseline STL generation

This step creates the STL for the accelerator with the smallest configuration possible (Baseline) by leveraging common AI primitives applications that naturally exercise the whole ensemble of FUs. First, we generate suitable input patterns (*test data*) to detect hardware faults in VPU units while executing these specific workloads. Secondly, we add some Ad Hoc code targeting hard-to-detect faults.

1) ***Input patterns exploration****:* For the first stage, we use RTL abstraction and experimentally explore software parameters, such as data format (e.g., 32, 16, or 8 bit integers), input values, and sizes, for matrix multiplication (MxM) and 2D Convolution (Conv) applications, aiming to find datasets that best excite and propagate faults without requiring changes in the code. In detail, randomized modifications of input data and systematic size variations are used to determine the amount of data required to achieve acceptable FC. FC is estimated for the VPU's FUs through single-fault RT-level *Fault Injection* (FI) campaigns (resorting to permanent SAFs on the wires and flip flops) using both application and pseudo-random data. In this analysis, we adopt a bottom-up strategy, so we start with a reduced fault space focused on individual FUs, e.g., targeting only FUs used in the application. By isolating these units, we could analyze their fault behavior in detail before expanding to more comprehensive FI campaigns targeting the whole VPU.

Once the FC is estimated on the target VPU units, the process is repeated to find the FC saturation point by increasing the size of the input test data using multiple pseudo-random seeds and iteratively doubling the data size, as illustrated in Algorithm 1. The process starts with a small dataset size (e.g., 2), and for each seed, we estimate the achieved FC (FC_{curr}). Then, the input data size increases, and new datasets are generated and evaluated. After each iteration, the FC is compared with that from the previous step (FC_{prev}). This process is repeated until the maximum FC value remains unchanged

Algorithm 1 Identification of Fault Detection Saturation Point

$FC_{prev} \leftarrow \emptyset$, $FC_{curr} \leftarrow \emptyset$, $Data_Size \leftarrow 2$
repeat
  $FC_{prev} \leftarrow FC_{curr}$
  $FC_{curr} \leftarrow \emptyset$
  for $seed$ in $\{0$ to $n-1\}$ **do** ▷ e.g., n = number of seeds
    $Data \leftarrow gen_data(Data_Size, seed)$
    $FC_{curr} \leftarrow \max(FC_{curr}, calc_FC(Data))$
  end for
  $Data_Size \leftarrow Data_Size * \Delta$ ▷ e.g., $\Delta = 2$
until $FC_{curr} = FC_{prev}$

TABLE I
CLASSIFICATION OF UFS AND TEST STRATEGIES

Classification	Cause	Test Strategy
Unused FU unit	The application does not exercise all FUs	Add instructions that explicitly target unused FUs
Unused instructions	VPU instructions not present in the test code for all the threads using the VPU	Add instructions in the test with pseudo-random operands for all the threads
Unexplored data representations	The application uses only one data representation operation, e.g., 32-bits	Generate input vectors with different data-width configurations
Insufficient data diversity	Limited operand variation, e.g, fixed-shift amount	Create further datasets to increase operand variability

between two different input sizes, indicating that additional data no longer contributes to increasing the coverage.

*2) **Ad Hoc routines generation**:* This step starts by performing a gate-level fault simulation of the whole baseline VPU with the code and data coming from the previous stage. Structurally safe faults are removed from the fault list.

Afterwards, Ad Hoc routines are added targeting still undetected faults, which can be categorized as summarized in Table I. For every category, the rightmost column details the strategy to test the corresponding faults with suitable code specifically designed to activate unused VPU units, such as idle FUs, unused data widths and formats, and unused instructions. This code employs carefully selected vector instructions to drive the hardware into functional states not reached by the 2 previously considered applications. In this case, we first identify where the UFs are through RTL code analysis. Then, for each category in the Table I, we generate suitable code fragments that are included as part of the Ad Hoc routines and used to ensure uniform activation over all the accelerator. In detail, instructions are added to exercise the unused parts of the VPU, with input data from the previous step, but with different operations and data representations.

Finally, we exploit the inherent multi-threading capability in VPUs to enhance test performance. We assign a dedicated thread to each of the two initially selected applications, enabling the concurrent execution of multiple workloads while testing several units. Once the Ad Hoc routines are generated, they are integrated into the system in a new thread. This approach provides a non-intrusive augmentation in the application's testing features, achieving a higher overall FC with low code overhead and minimal impact on the STL code.

B. STL customization

In this step, when the baseline STL is ready and provides acceptable coverage levels, the **PIPPO** framework generates the final STL according to the target HW parameters and its configuration (e.g., number of parallel cores and FU/lanes). This is done by modifying the input data and code of the test, by replicating input vectors and instructions to stimulate all the FUs and lanes uniformly. Specifically, the test vectors are modified to match the size of the FUs, replicating the data to ensure that the crafted inputs stimulate all instances with the same patterns. The data used in the applications is increased in size to maintain the number of operations per FUs at a level superior to or equal to the baseline configuration. Finally, if the VPU features multiple lanes, the baseline STL is replicated to leverage the hardware's IMT, which allows the same workload to be executed concurrently across all lanes, mapping each thread to a distinct lane.

IV. EXPERIMENTAL SETUP

For the RTL experiments, we built a custom framework that integrates a logic simulator (Modelsim by Siemens EDA) with a fault injector infrastructure and a Python script controller to manage the fault injection campaigns on the target RTL-level VPU model. Another Python script is responsible for generating the fault list from a list of RTL signals and parameters. In the experiments, we injected a single permanent stuck-at fault per simulation at the RT level. Then, the framework compared the final memory content with the fault-free version.

At the end of each simulation, the faults are classified into: *i)* ***Detected*** (DF) when the fault is detected by mismatch between the golden model, *ii)* ***Undetected Fault*** (UF) when the faulty-simulated results are equal to the fault-free reference. Moreover, UFs are further divided into two categories: *i)* ***Safe Faults*** (SFs), comprising faults that can not affect system results no matter the executed instructions and applied stimuli; and *ii)* ***Unsafe Faults*** (USFs), which are faults that fail to propagate to observable states and could possibly affect the system behavior if the dataset and/or the instruction flow changes. Finally, the FC is calculated as the ratio of DFs to the total number of faults excluding the safe ones.

To quantify at the GL the effectiveness of the STLs, we resorted to some fault injection campaigns using the Z01X functional fault simulator by Synopsys after synthesis with *Design Compiler* by Synopsys and a 15nm PDK [29]. We used a formal verification tool (*JasperGold* by Cadence) to analyze and identify safe faults in the VPU units through formal structural activation and propagation analyses.

Finally, we implemented the **PIPPO** framework in Python, consisting of approximately 850 lines of code. It receives as input the baseline HW parameters and the target configuration for which the STL must be generated, the C code from the baseline STL, and the input data, and generates the final STL for the target hardware. Internally, it first parses the STL code and data to identify the vector instructions and operands. Afterward, it processes the HW parameters to calculate the new vector sizes and finally generates the final STL.

V. Experimental results

To validate the proposed methodology, we first evaluated the *Klessydra-T* VPU in three hardware configurations (*Baseline, or 'STD'*, *Extended SIMD, or 'eSIMD'*, and *Symmetric, or 'SYM'*). Table II reports the fault spaces corresponding to the targeted configurations at the RT and gate level.

Then, for validation purposes, we extended our evaluation and analysis to six VPU configurations that allow us to understand how scaling the HW impacts the VPU reliability by varying a single parameter at a time from the baseline configuration, starting with the number of FUs per lane and then the number of vector lanes.

For the RTL experiments, several FI campaigns were first performed across the three VPU configurations, targeting the detection of permanent SAFs in the registers and wires of the FUs. Moreover, two key applications (MxM and Conv) for AI algorithms served as the basis for the STL's development. In detail, the exploration of data patterns in MxM and Conv applications began with 2x2 pseudo-random input matrices and progressed to 64×64 matrices, at which point the saturation point was detected for the eSIMD configuration, and 16x16 for the other two configurations. Conv used one input image operated with five fundamental image processing 3×3 kernels [30]: *1)* Identity, *2)* Horizontal Sobel, *3)* Vertical Sobel, *4)* Box blur, and *5)* Sharpen. The images were generated with two distinct data ranges: a fixed-range input (ranging between $\pm 1,024$) and a fully randomized input, with sizes from 2×2 to 64×64. For MxM and Conv, the pseudo-random test datasets were generated 3 times with different seeds.

The saturation point was achieved after 36 iterations: the increase in RTL FC during Algorithm 1 execution between the starting and final point accounted for about 77%, showing the effectiveness of the approach. After that, the gate-level FC was computed for each configuration, using *zoix* to inject stuck-at faults in the MFU and measure the FC achieved by each application (Convolution and MxM), by the Ad Hoc routines, and by the final STL.

All the experiments were performed on an Intel(R) Xeon(R) Gold 6238R CPU @2.20GHz with 252 GB of RAM. A single FI campaign at the RT level required around 3 h to complete, while each gate-level FI required around 12.4 h. Instead, the **PIPPO** execution time required a few seconds per STL.

While other configurations are mainly considered to demonstrate the framework's scalability, the STL development initially focuses on the STD configuration, due to its streamlined design and faster simulation times, enabling detailed FC evaluation and trade-offs across workloads.

To minimize the required test time, the MxM, Conv, and Ad Hoc codes are executed concurrently, each running in a separate thread. To further improve FC by increasing data representation diversity, MxM was configured with 16-bit (16b) data and Conv with 32-bit (32b) data.

Table III reports the Memory Footprint (MF), Instruction Count (IC), Cycle Count (CC), and gate-level stuck-at FC for the different parts of the STLs targeting each VPU configuration. When computing the FC, we removed from the fault list the structurally safe faults identified via JasperGold for the STD (1,144 faults), eSIMD (3,889), and SYM (3,410) configuration. The results are presented for the single applications (MxM, Conv, and the Ad Hoc routines), and the final multi-thread STL (combining Conv, MxM, and the Ad Hoc routines).

TABLE II
Evaluated VPU configurations and their Fault Spaces

Klessydra configuration			Fault Space	
			RTL	GL
Name	Lanes	FUs per Lane	FUs (FFs & Wires)	SAFs
Baseline (*STD*)	1	2	3,472	81,098
Extended SIMD (*eSIMD*)	1	8	12,712	273,295
Symmetric (*SYM*)	3	2	10,376	237,192

TABLE III
STL features for the evaluated VPU configurations

VPU	STL	MF (Byte)	IC	CC	FC
STD	MxM	2,048	1,427	3,854	45.33%
	Conv	1,204	1,627	25,631	68.23%
	Ad Hoc routines	568	2,521	17,052	93.65%
	Baseline STL	3,308	2,838	57,027	**94.29%**
eSIMD	MxM	32,768	1,427	82,363	47.21%
	Conv	16,564	1,627	76,567	68.15%
	Ad Hoc routines	2,272	2,532	46,093	92.99%
	Final STL	49,332	2,841	221,108	**96,26%**
	Baseline STL	3,308	2,838	51,481	94.68%
SYM	MxM	2,048	1,396	4,081	46.02%
	Conv	1,204	1,654	20,592	67.92%
	Ad Hoc routines	568	1,979	15,623	92.16%
	Final STL	3,308	2,269	42,260	**92.83%**
	Baseline STL	3,308	2,838	55,129	68.28%

To quantify the impact of **PIPPO**, we also evaluated the baseline STL without any adaptation on the *eSIMD* and *SYM* configurations. Interestingly, we observe a FC drop of about 25% (from about 68% to nearly 92%) for the *SYM* configuration between the baseline STL and the one generated by **PIPPO**. This highlights the need to modify the original STL to fit variations in the HW configuration.

For the STD configuration, the final STL achieved a 94.29% FC. Where the remaining 5.71% of faults are safe faults that could not be identified by EDA tools and can be classified as such via manual analysis. Comparing the FC only for the FUs the results differed by less than 3% from the RTL estimation, showing a good correlation with the GL experiments.

The duration of each STL (CC) takes into account the time for uploading the input matrices into the VPU. The final STL for each configuration derives from the 3 composing parts sharing some common instructions. Since STLs often run during the application's idle times, the CC should be as short as possible. In case the STL duration is too high, it can be split into chunks fitting the available idle time.

While the applications alone achieve acceptable levels of FC for FUs, they miss many hard-to-detect faults, and faults in the control logic or hardware parts not used by the program. Our Ad Hoc routines are specifically designed to target these gaps. Indeed, experimental results for the *STD* configuration confirm that the achieved FC is close to the final combined STL (within 1%). This is expected, as this HW configuration has few FUs. Hence, the instructions targeting the control logic also enable testing to stimulate a significant portion of the FUs.

The value of using the two applications as a starting point

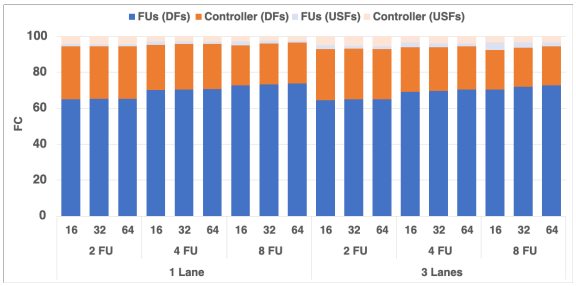


Fig. 3. Gate-level FC for different VPU configurations and data sizes

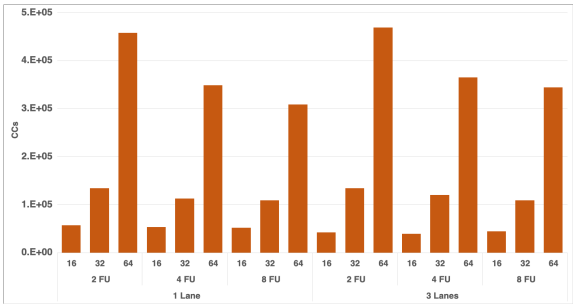


Fig. 4. CCs for different VPU configurations and data sizes

becomes evident when the number of FUs per lane becomes larger (i.e., in the eSIMD configuration), as shown in Table III. As the number of FUs increases, these applications provide a highly efficient method for exercising the many replicated FUs, while the Ad Hoc codes remain crucial for the non-scalable control logic. Despite the high final FC achieved, the Ad Hoc routines present a moderate cost in terms of IC, consisting of a set of crafted instructions following the strategies in Table I. It is worth noting that these routines reuse the same test data as the applications. Thus, when executed with the main applications, they introduce minimal overhead in terms of MF.

Then, to further validate the approach on a larger set of configurations, the STLs generated for them, by the **PIPPO** framework, were evaluated in a post-synthesis scenario through comprehensive gate-level FI campaigns. The evaluation covered multiple VPU configurations: 1 and 3 lanes, each having 2, 4, and 8 FUs per lane. Figure 3 reports the FC results for this set of configurations and three different input data sizes, detailing the distribution of DFs and USFs between the FUs and the controller. In addition, Figure 4 reports the CCs for the same set of configurations and input data size.

Specifically, we observe that larger test data sizes lead to higher FC until a saturation point is reached, confirming the results obtained in the RTL simulations phase. Notably, the required size to reach this FC saturation scales with the number of FUs per lane, requiring 16x16 inputs for 2 FUs, 32x32 for 4 FUs, and 64x64 for 8 FUs per lane. This highlights the importance of choosing a suitable test data size to achieve the highest FC (up to 96%) while avoiding unnecessary execution overhead when processing larger matrices that do not contribute to the overall FC. Complementary analyses of the FC and CC results also enable a trade-off between FC and test duration: although larger data sizes achieve higher FC (up to the saturation point), they also result in longer execution times, e.g., in a VPU configuration with 1 lane and 4 FUs, the CCs increase 6.5× when moving from 16x16 to 64x64 under minimal FC improvement (from 95.11% to 95.66%).

From the VPU's architectural point of view, we noticed that increasing the number of FUs per lane allows for achieving a higher overall FC. In particular, the number of USFs (concentrated in the control logic) remains mostly constant in larger VPUs, whereas the FUs (targeted by most test patterns) are replicated and become the dominant factor in the FC calculation. In contrast, increasing the number of lanes leads to a larger and more complex control path, featuring additional multi-lane logic. Thus, FUs might become a smaller portion of the total fault space w.r.t. the control logic and decrease the overall FC. However, increasing the number of FUs per lane counteracts this effect, raising the FC by increasing the FUs area.

The FC breakdown further supports these results, showing that the FC of the FUs reaches approximately 98%, whereas the control logic FC consistently achieves around 87% across all configurations. This demonstrates that a critical aspect in the development of STLs for VPUs relies mainly on testing the FUs, as their large area drives the overall FC.

VI. Conclusion

This work introduced a software-based strategy for the in-field detection of permanent faults in VPUs. Our method combines application code reuse, generation of suitable input test data, multi-threaded execution, and the development of Ad Hoc routines. The approach first uses the RTL description of a VPU to effectively explore and evaluate software parameters, including input test data and sizes. Then moves to GL and generates suitable Ad Hoc routines addressing the hard-to-test faults, leading to an effective baseline STL. Finally, our **PIPPO** framework automatically customizes the baseline STL for different, more complex hardware configurations.

The experimental results demonstrate that our strategy achieves remarkable fault coverage (over 96% for the eSIMD configuration) with moderate test time, zero area overhead, and without using any ATPG tool. Overall, the proposed methodology is particularly attractive for resource-constrained systems, where strict design budgets prohibit architectural hardware changes to support DfT (e.g., BIST). In addition, the results show that the method allows for selecting the best trade-off between the FC and the required test time.

Furthermore, working at the RT level for the first evaluation phase allows us to speed up the STL development when compared to GL evaluations, enhancing the capability to explore the possible data sets. This reduces the preparation time for each experiment and enables faster iteration cycles, leading to quick fault space exploration within a limited budget.

In future works, we plan to extend the method to other accelerators and fault models (e.g., delay ones with timing-aware Ad Hoc routines). Furthermore, we will investigate test performance optimization by exploring test compaction to maximize FC and reduce execution time.

REFERENCES

[1] B. Dally, "Hardware for Deep Learning ," in *2023 IEEE Hot Chips 35 Symp. (HCS)*, 2023, pp. 1–58.

[2] W. A. Nunes *et al.*, "Accelerating machine learning with risc-v vector extension and auto-vectorization techniques," in *IEEE Int. Symp. on Circuits and Systems (ISCAS)*, 2025, pp. 1–5.

[3] S. Matsuoka, "Fugaku and a64fx: the first exascale supercomputer and its innovative arm cpu," in *Symp. on VLSI Circuits*, 2021, pp. 1–3.

[4] M. Sato *et al.*, "Co-design for a64fx manycore processor and "fugaku"," in *Int. Conf. for High Performance Computing, Networking, Storage and Analysis (SC'20)*, 2020, pp. 1–15.

[5] A. Cheikh, *et al.*, "Klessydra-t: designing vector coprocessors for multithreaded edge-computing cores," *IEEE Micro*, vol. 41, no. 2, pp. 64–71, 2021.

[6] M. Valero and R. Espasa, "Multithreaded Vector Architectures," in *2013 IEEE 19th Int. Symp. on High Performance Computer Architecture (HPCA)*, 1997, p. 237.

[7] A. Chaudhuri, J. Talukdar, and K. Chakrabarty, "Special session: Fault criticality assessment in ai accelerators," in *2022 IEEE 40th VLSI Test Symposium (VTS)*, 2022, pp. 1–4.

[8] A. Chaudhuri, J. Talukdar, F. Su, and K. Chakrabarty, "Functional criticality analysis of structural faults in ai accelerators," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 12, pp. 5657–5670, 2022.

[9] A. J. Strojwas *et al.*, "Yield and reliability challenges at 7nm and below," in *Electron Devices Technology and Manufacturing Conf. (EDTM)*, 2019, pp. 179–181.

[10] "The International Roadmap for Devices and Systems: A Beacon for the Electronics Industry," *Computer*, vol. 55, no. 08, pp. 4–4, Aug. 2022. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/MC.2022.3174253

[11] H. D. Dixit, S. Pendharkar, M. Beadon, C. Mason, T. Chakravarthy, B. Muthiah, and S. Sankar, "Silent data corruptions at scale," 2021. [Online]. Available: https://arxiv.org/abs/2102.11245

[12] H. Lee *et al.*, "Low-overhead aging-aware resource management on embedded gpus," in *54th Ann. Design Automation Conf. (DAC)*, 2017.

[13] P. Bernardi *et al.*, "Development flow for on-line core self-test of automotive microcontrollers," *IEEE Trans. Comput.*, vol. 65, no. 3, pp. 744–754, 2016.

[14] J. E. R. Condia *et al.*, "Using stls for effective in-field test of gpus," *IEEE Design & Test*, vol. 40, no. 2, pp. 109–117, 2023.

[15] N. Karystinos, G.-M. Fragkoulis, O. Chatzopoulos, D. Gizopoulos, and S. Gurumurthi, "Harpocrates++: Automated functional program generation against cpu faults and silent data corruptions," *IEEE Micro*, pp. 1–9, 2025.

[16] A. Ruospo *et al.*, "Image test libraries for the on-line self-test of functional units in gpus running cnns," in *IEEE European Test Symp. (ETS)*, 2023.

[17] R. Limas Sierra *et al.*, "A structured method to generate self-test libraries for tensor cores," *Electronics*, vol. 14, no. 11, 2025. [Online]. Available: https://www.mdpi.com/2079-9292/14/11/2148

[18] J. Yu *et al.*, "Vector processing as a soft processor accelerator," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 2, no. 2, 2009.

[19] M. Perotti *et al.*, "A "new ara" for vector computing: an open source highly efficient risc-v v 1.0 vector processor design," in *Int. Conf. on Application-specific Systems, Architectures and Processors*, 2022, pp. 43–51.

[20] F. Minervini *et al.*, "Vitruvius+: An area-efficient risc-v decoupled vector coprocessor for high performance computing applications," *ACM Trans. Archit. Code Optim.*, vol. 20, no. 2, 2023.

[21] S. Liu *et al.*, "Toward full-stack acceleration of deep convolutional neural networks on fpgas," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 33, no. 8, pp. 3974–3987, 2022.

[22] M. Barbirotta *et al.*, "Enhancing fault tolerance in high-performance computing: A real hardware case study on a risc-v vector processing unit," *IEEE Open Journal of the Computer Society*, vol. 5, pp. 553–565, 2024.

[23] M. Barbirotta, A. Cheikh *et al.*, "Fault-tolerant hardware acceleration for high-performance edge-computing nodes," *Electronics*, vol. 12, no. 17, 2023. [Online]. Available: https://www.mdpi.com/2079-9292/12/17/3574

[24] M. Barbirotta *et al.*, "Analysis of a fault tolerant edge-computing microarchitecture exploiting vector acceleration," in *Conf. on Ph.D Research in Microelectronics and Electronics*, 2022, pp. 237–240.

[25] M. Barbirotta, F. Menichelli *et al.*, "Dynamic triple modular redundancy in interleaved hardware threads: An alternative solution to lockstep multi-cores for fault-tolerant systems," *IEEE Access*, vol. 12, pp. 95 720–95 735, 2024.

[26] S. Hukerikar *et al.*, "Optimizing large-scale fault injection experiments through martingale hypothesis: a systematic approach for reliability assessment of safety-critical systems," in *54th Ann. IEEE/IFIP Int. Conf. on Dependable Systems and Networks - Supplemental Volume (DSN-S)*, 2024.

[27] E. Sanchez and M. Sonza Reorda, "On the functional test of branch prediction units," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, 2015.

[28] F. Corno *et al.*, "Automatic test program generation: a case study," *IEEE Design & Test of Computers*, vol. 21, no. 2, pp. 102–109, 2004.

[29] M. Martins *et al.*, "Open cell library in 15nm freepdk technology," in *Proceedings of the 2015 Symposium on International Symposium on Physical Design*, ser. ISPD '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 171–178. [Online]. Available: https://doi.org/10.1145/2717764.2717783

[30] Z. Li *et al.*, "A survey of convolutional neural networks: Analysis, applications, and prospects," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 33, no. 12, pp. 6999–7019, 2022.