Reliability and Performance Evaluation of a RISC-V Vector Extension Unit for Vector Multiplication

Carolina Imianosky, Douglas A Santos, Douglas R Melo, Felipe Viel, Luigi Dilillo

Title: Reliability and Performance Evaluation of a RISC-V Vector Extension Unit for Vector Multiplication

Author(s): C. Imianosky, D. A. Santos, D. R. Melo, F. Viel, and L. Dilillo

Document version: Pre-print version (Final draft)

Please cite the original version:
C. Imianosky, D. A. Santos, D. R. Melo, F. Viel, and L. Dilillo, "Reliability and Performance Evaluation of a RISC-V Vector Extension Unit for Vector Multiplication, "2024 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2024.

Reliability and Performance Evaluation of a RISC-V Vector Extension Unit for Vector Multiplication

Carolina Imianosky*, Douglas A. Santos*, Douglas R. Melo†, Felipe Viel†‡, Luigi Dilillo*
*IES, University of Montpellier, CNRS, Montpellier, France †LEDS, University of Vale do Itajaí, Itajaí, Brazil
‡SpaceLab, Federal University of Santa Catarina, Florianópolis, Brazil
{carolina.imianosky, douglas.santos}@umontpellier.fr, {drm, viel}@univali.br, luigi.dilillo@umontpellier.fr

Abstract

The RISC-V Vector Extension (RVVE) enhances computational efficiency by exploring data-level parallelism, which can benefit Artificial Intelligence (AI) applications. The Zve32x subset is tailored for embedded systems, including those in space, in which AI applications are very useful to process data onboard. However, environments like space challenge the reliable functioning of electronic systems due to radiation exposure and extreme temperatures. Thus, fault tolerance techniques are crucial to mitigate potential failures. Therefore, we extended a previous implementation of a subset of the RVVE to the HARV-SoC, a fault-tolerant RISC-V-based system-on-chip, to add support to multiplication operations, which are essential in AI applications. We also added support to more configuration instructions that allow increasing the number of elements that are processed with a single instruction, further accelerating the processing. We evaluated the impact of the vector instructions over the scalar ones using C and Assembly applications to evaluate the impact of vector intrinsic functions. We estimated the reliability of the core through simulations based on fault injection campaigns with both baseline and hardened HARV-SoC. The results show that the vector unit offers a performance acceleration of up to 28.69 times with manual code optimization and can enhance reliability. Furthermore, this work assesses how different configurations of the vector parameters affect the performance and reliability.

Index Terms

RISC-V, System-on-Chip, Vector Instructions, Fault Tolerance, Space Systems

I. Introduction

The RISC-V is an open standard Instruction Set Architecture (ISA) that is simple and modular, enabling processor implementations to be designed to meet specific needs. The RISC-V Vector Extension (RVVE) is one of the modules available, and it enables systems to execute a single instruction on multiple data elements simultaneously, exploring Data-Level Parallelism (DLP). The ratified version 1.0 of the RVVE is specified in [1].

Since RVVE is the largest RISC-V extension [1], its full implementation can be costly regarding the area and power consumption, which is not ideal for embedded systems. To address this issue, the RISC-V committee introduced the Zve* extensions, consisting of five subsets of RVVE. These extensions limit the vector operations to 32-bit and 64-bit integers, fixed-point, and single- and double-precision floating-point. Notably, the Zve32x restricts the element width to 32 bits and supports only integer data. Thus, it is possible to speed up image and video processing applications, such as those based on Artificial Intelligence (AI) [2], in embedded systems where low latency and power efficiency are crucial, as in space applications.

AI can be used in many space applications, such as remote sensing for image classification and target detection [3]. Additionally, AI can be used to process data onboard nano- and pico-satellites instead of sending them to the ground station, which reduces the demand for downlink communication and increases capabilities and dependability [4].

However, the harshness of the space environment imposes challenges on the design of dependable systems. Exposure to radiation, extreme temperatures, and several other conditions can affect the operation of electronics, leading to critical failures. Thus, applying fault tolerance techniques in these systems is crucial to meet reliability constraints [5].

The results presented in this paper have been obtained in the framework of the EU project RADNEXT, receiving funding from the European Union's Horizon 2020 research and innovation programme, Grant Agreement no. 101008126, the Region d'Occitanie and the École Doctorale I2S from the University of Montpellier (contract no. 00137932/22009671), the Foundation for Support of Research and Innovation, Santa Catarina (FAPESC-2021TR001907), the Brazilian National Council for Scientific and Technological Development (CNPq - processes 408641/2023-1 and 350794/2023-5), and Project HARV (project PE24PR01) in the framework of the action "Accelerateur d'innovation" from the University of Montpellier.

HARV (HArdened RISC-V) [6] is a low-cost fault-tolerant RISC-V processor. It is hardened against SEUs (Single-Event Upsets) and SETs (Single-Event Transients) using Triple Modular Redundancy (TMR) and Error-Correcting Code (ECC) and has been characterized in different irradiation facilities [7], proving suitable for reliable applications.

In our previous work [8], we presented a Vector Extension Unit (VEU) for the HARV processor, supporting a limited subset of the vector instructions: sequential memory access, basic vector configurations, and integer arithmetic operation ($addition$, $subtraction$, and, and xor). However, it lacked support for the multiplication operation, which is crucial for AI algorithms [9], [10].

This work is based on the Zve32x and presents the improvement of the VEU proposed in [8]. We added support to vector multiplication operations and additional required vector configuration instructions. We also added support for the LMUL parameter, which groups the vector registers, allowing for fewer but longer vectors. This approach allows for a more cost-effective hardware implementation while supporting vectors of the same length as those in [8].

We extended the validation by comparing benchmarks using scalar and vector instructions in HARV with both baseline and hardened configurations. Additionally, this work presents a reliability characterization of the VEU in the HARV System-on-Chip (HARV-SoC), performed through fault injection simulations. We assessed different configurations of the VEU parameters to analyze how they impact the performance and reliability of applications that execute vector multiplications. This assessment provides a valuable analysis of the impact of using vector extensions for reliable applications.

The remainder of this paper is organized as follows. Section II discusses the related work. Next, Section III presents additional concepts on the RISC-V ISA and its vector extension, and Section IV describes our VEU design. Section V presents the experimental methodology employed in this study, while Section VI discusses the experimental results. Finally, the final remarks are shown in Section VII.

II. Related Work

Some works in the literature have implemented the RVVE for embedded processors. For example, [11] describes a minimalist implementation of the RVVE aimed at embedded devices. Another example is the [12], which presents the design and implementation of a pluggable vector unit. This research extends the open-source RISC-V processor CAV6 to support the RVVE, specifically targeting the Zve64x extension.

In the context of reliable and space applications, [13] details the implementation of the RVVE on the HPP64 NOEL-V platform. This work aims to enhance the performance and capabilities of space processors while considering the specific constraints and requirements of satellite data systems. Additionally, the authors of [14] implemented fault tolerance techniques in the Vitruvius+ architecture, a RISC-V vector coprocessor for High-Performance Computing (HPC), to address the growing complexity and error vulnerability in modern HPC.

While [14] presents fault injection simulation results, it is not focused on embedded systems. Similarly, [13] is designed for space applications, utilizing a hardened processor core, but does not provide a reliability evaluation. In contrast, the works by [11] and [12] focus on embedded systems but do not address reliability or fault tolerance. Among the metrics analyzed in each implementation, operating frequency, performance, and hardware overhead are consistently evaluated. Besides that, all the mentioned works support multiplication operations, which is essential for AI applications [10], [15].

Given this context, we improved the implementation from [8], supporting multiplication operations and targeting the Zve32x extension. We assessed the performance gain and hardware overhead of the implementation. Additionally, we performed fault injection simulations to conduct a comprehensive reliability analysis.

III. RISC-V Vector Extension

Vector architectures originated from SIMD processing, which partitions large registers into multiple elements and operates them in parallel. In SIMD instructions, the operation and data width are defined by the operation code, i.e., increasing the length of the vectors also implies an increase in the instruction set [16]. On the other hand, in vector architectures, the implementation defines the data width. Thus, it reduces the size of the instruction set and makes the hardware design more flexible for data parallelization without affecting algorithm development.

The RVVE specifies the ELEN and VLEN parameters. The ELEN parameter represents the maximum size in bits of a vector element. The VLEN indicates the number of bits in each of the 32 vector registers in the Vector Register File (VRF) specified by the RVVE. Also, seven Control and Status Registers (CSRs) are added to support the RVVE [17], and their values are modified by the Configuration-Setting instructions. The instruction *setvl* needs to be the first vector instruction to be executed since it sets three mandatory parameters: Vector Lenght (VL), vector Lenght Multiplier (LMUL), and the Selected Element Width (SEW).

The VL denotes the number of elements a specific instruction will operate on, and it can be a value between 1 and VLMAX (Equation 1). The LMUL is a parameter that can change the granularity of the register file since it allows multiple

vector registers to be grouped and operated on as one vector. The SEW represents the width of the elements and acts as a divider, breaking down the vector register into multiple elements of the specified width [1].

$$\text{VLMAX} = \frac{\text{LMUL} \times \text{VLEN}}{\text{SEW}} \tag{1}$$

IV. HARV Vector Extension Unit

This work proposes an improved architecture implementation introduced in [8] but based on the Zve32x. Thus, this VEU sets the following parameters: (i) ELEN is set to 32 bits, i.e., supports element widths of 8, 16, and 32 bits; and (ii) VLEN is set to 32 bits, i.e., the registers in the VRF are 32-bit wide.

In this work, we extended the supported subset of vector instructions. Thus, the supported instructions include (i) all vector configuration instructions, (ii) unit-stride load and store instructions, (iii) integer arithmetic operations ($addition$, $subtraction$, $multiplication$, AND, XOR, and OR). We also provided support for setting the LMUL parameter, allowing it to have up to 256-bit vectors (when LMUL = 8) even with the VLEN set to 32. With this, it is possible to decrease the overhead of logic resources while allowing the same maximum vector length as the old implementation.

The Fig. 1 shows an overview of the VEU. For this implementation, we separated the vector control from the main control unit of the core. Thus, the VEU comprises the control unit, Vector Register File (VRF), elements counter, execution unit, and interface unit.

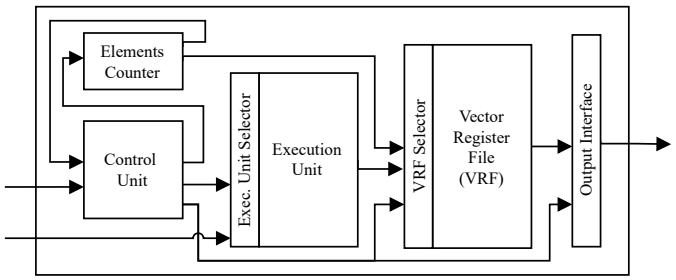

Fig. 1. Vector Extension Unit

The elements counter holds the number of vector elements that were already processed. Signals from the vector control unit specify the value and when the counter should be incremented. The Interface Unit is responsible for directing the outputs of the VEU. The outputs are a signal indicating if all the necessary elements were processed (*v_done*) and two arrays of data that serve for memory access.

The main control unit of the core integrates the instruction decoder. The instruction bits 6 to 0 are the input to a combinational logic component that determines the instruction format, according to the RISC-V ISA specification [16]. Afterward, an output of the control unit that signals if it is a vector instruction *v_inst* is either enabled (1) or disabled (0). In the case of a vector instruction, it waits for the signal *v_done* to proceed with the execution.

The vector control unit has as inputs the instruction to be executed, the values from the vector CSRs, and the *v_inst*, that is used to initiate the decode and execution of the vector instructions. To apply the same logic as the HARV microarchitecture [6], the vector ALUs are controlled by the vector control unit to simplify the implementation.

The execution unit, represented in Fig. 2, comprises one 32-, one 16-, and two 8-bit ALUs. These ALUs are simplified and exclusive to arithmetic vector instructions, operating only on the vector elements. With VLEN being 32 bits, it can perform either one 32-bit, two 16-bit, or four 8-bit operations simultaneously. Therefore, the level of parallelization of operations depends on the SEW value. The inputs and outputs are mapped to their respective ALUs according to the SEW value. The elements that are being operated on wider ALUs are zero-extended.

The VRF contains 32 VLEN-bit registers exclusive to the VEU and can perform two readings and one writing simultaneously. The data outputs are defined to have the size of VLEN so that all the vector elements are read at once. When LMUL $>$ 1, adjacent vector registers are merged to create register groups, i.e., a register file with fewer but longer vector registers.

V. Experimental Methodology

In this section, we describe the experimental methodology applied to this work. The following subsections detail the development and synthesis tools, the benchmark algorithms, and the performance and reliability evaluation.

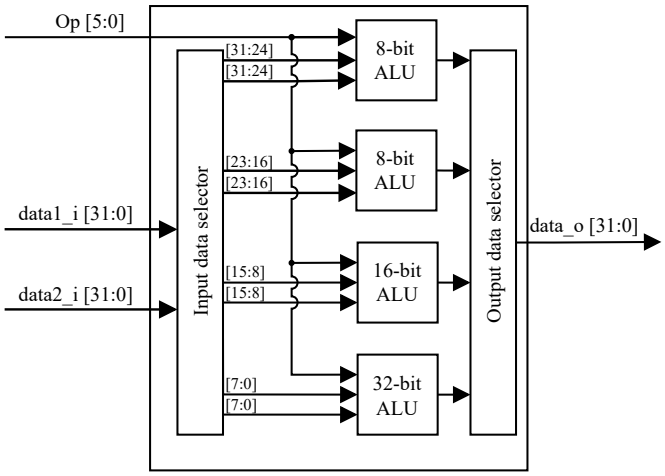

Fig. 2. Vector Execution Unit

A. Development and Synthesis Tools

The VEU was implemented using VHDL (VHSIC Hardware Description Language) since HARV-SoC was also implemented using this HDL. We used TCL (Tool Command Language) and Python scripts with the ModelSim SE-64 2019.01 software for the fault injection simulations and reliability evaluation.

All the algorithms were implemented in C or Assembly language. For the vector instructions in C, we used the RISC-V vector C intrinsics [18]. We compiled the applications using the GNU Compiler Collection (GCC) for RISC-V [19].

The synthesis data were collected using the Xilinx Vivado 2020.2 Design Suite tool and the Zynq ZC7020 FPGA SoC device. Thus, we analyzed the number of Look-up Tables (LUTs), Flip-Flops (FFs), and digital signal processing (DSP) blocks, as well as the maximum operating frequency (F_{max}), and the dynamic power dissipation (P_{dyn}) for the HARV-SoC.

B. Benchmark algorithms

We implemented an algorithm that reads elements of two vectors, multiplies them, and stores the result in memory. The number and size of the elements varied depending on the scenario that the algorithm was being executed. The benchmark algorithm was implemented in both scalar and vector versions using C language. Additionally, we implemented the scalar and vector versions in assembly language versions to maximize performance through low-level optimizations.

C. Performance evaluation

For the performance evaluation, we executed the benchmark algorithms to evaluate the performance variation using the baseline HARV (RV32I) and the HARV with VEU (RV32IZve32x), comparing both the C and assembly versions of the benchmark algorithm. This comparison allowed us to assess the impact of C and intrinsic functions on performance across different configurations. In all scenarios, the number of elements in each vector was set to 256. In the baseline HARV scenario, we varied the size of the elements between 8, 16, and 32 bits. For the executions with the VEU, we varied the LMUL parameter between 1, 2, 4, and 8 and the SEW between 8, 16, and 32 bits, with the VL parameter always being set to VLMAX.

We read the cycle counter register (*mcycle*) at the start and the end of the algorithm execution to measure the processing latency in both executions. The performance metric was obtained by dividing the number of cycles for execution in the baseline HARV by the number of processor cycles used for the executions with the VEU.

D. Reliability evaluation

We ran 1024 experiments for each of the four scenarios: baseline HARV-SoC, baseline HARV-SoC with the VEU, hardened HARV-SoC, and hardened HARV-SoC with the VEU. This way, we could assess the impact of the VEU in terms of fault tolerance for each SoC version.

We used the C language version of the benchmark algorithm for the reliability evaluation since it is more commonly used in practical applications. The number of elements in each vector was set to 16384. In the executions with the VEU, we set the LMUL parameter to 8 and the VL to VLMAX, aiming to exploit maximum parallelism. We varied the SEW

between the best and worst parallelization levels, i.e., 8 and 32 bits, to explore how the different SEWs impact reliability. Thus, we also operated on 8- and 32-bit elements in the scalar scenarios.

At the end of each execution, the results stored in the memory are verified to check if they match the expected multiplication result. If not, a message is printed to inform that the execution experienced an error. Each execution creates a log file with all the UART output and the simulation time.

The simulations start by executing a golden run, i.e., executing the application without fault injections. This run provides reference parameters for comparison with the subsequent simulations, such as execution time, UART output, and all the FFs of the implementation. These indicators are also used as parameters for the fault injections. When the golden run is finished, we run the fault injection simulations.

For each simulation, we run a script that randomly selects locations and times to inject bit flips based on neutron-characterized flip-flop FIT_{NYC} (Failure In Time for a billion hours in the New York City's neutron flux at sea level) of 248 and a neutron flux of $5\times10^{11} n/cm^2/s$. This value of flux was set after a set of preliminary simulation runs in order to ensure statistically meaningful data. At first, the script fetches all register signals and sets the simulation time to zero. Then, it increments the simulation time by a random value and calculates the neutron fluence from the previous simulation time until the incremented time. The script uses the cross-section and calculates the FF error rate for a given fluence. After this, it iterates through all FFs and randomly decides whether it should inject an error based on the error rate. Finally, it calculates the random injection time between the previous simulation and the incremented time and stores this in a file with the corresponding FF signal. This flow runs in a loop, incrementing the simulation time until it reaches the set maximum simulation time.

We evaluated the reliability of the SoC by analyzing the internal registers of the processor and the outputs. The executions were classified into four categories: (*i*) *correct*, when the application was executed correctly; (*ii*) *benchmark error*, when the application is executed but has incorrect results; (*iii*) *hang*, when the application does not end the execution nor produces any result; (*iv*) *uncorrectable*, when it detected an uncorrectable error; and (*v*) *exception*, when an unscheduled event disrupts the program execution [20]. The considered exceptions were instruction address misaligned, instruction access fault, illegal instructions, load access fault, and store access fault.

To estimate the trade-off between performance and reliability, we analyzed the metrics Mean Work to Failure (MWTF) and the event cross-section. We considered as an event all the cases in which the execution was not classified as correct. Based on [21], we calculated the MWTF by Equation 2, where a unit of work is considered one execution of the algorithm, thus the amount of work is 1024. The cross-section (XS) is a calculation of the error rate of the device based on the fluence expressed in cm^2/device.

$$\text{MWTF} = \frac{\text{amount of work}}{\text{number of errors encountered}} \tag{2}$$

We used a Python script to analyze the files containing the outputs of each execution and determine the number of executions for each category, the MWTF, the event XS, the average number of injected faults, and the execution time.

VI. Results

The following subsections introduce the results of the VEU implementation. First, we present the synthesis results, followed by an analysis of the performance achieved compared to the baseline processor. Finally, we show the results of the fault injection simulations.

A. Synthesis

Table I shows the resource usage for the HARV-SoC and the HARV-SoC with the VEU in both baseline and hardened scenarios. The table presents the number of FFs, LUTs, DSPs, maximum operating frequency, and dynamic power dissipation for each implementation.

TABLE I
Synthesis results.

HARV-SoC	LUTS	FFs	DSP	F_{max} (MHz)	P_{dyn} (mW)
Baseline	3748	3583	4	65.40	138
Baseline with VEU	5399	4170	9	64.40	140
Hardened	5802	4381	12	41.91	164
Hardened with VEU	7555	4968	17	40.77	175

In the baseline scenario, adding the VEU increases the number of LUTs by 44%. The number of FFs increases by 16%, mainly because of the VRF, and the number of DSPs increases due to the vector execution unit. Using fault tolerance techniques increases the number of logical resources the processor uses. The number of DSPs in the hardened scenario increased 3 times compared to the baseline because of the TMR implemented in the ALU. Adding the VEU to the hardened system increases LUT usage by 30% and FFs by 13%.

The maximum operating frequency of the processor decreases as the critical path increases. In the baseline implementation, the SoC can operate at a maximum of 65.40 MHz, and it drops 35% when applying the hardening techniques. Adding the VEU slightly decreases F_{max} by 1.5% in both baseline and hardened scenarios. The addition of the VEU also results in a moderate increase in P_{dyn} by 1.4% and 6.7% in the baseline and hardened scenarios, respectively.

B. Performance

The plot in Fig. 3 presents the variation in the performance achieved by different configurations of the benchmark algorithm relative to the scalar C version. We varied the LMUL parameter between 1, 2, 4, and 8 and the SEW between 8, 16, and 32. The VL parameter was set to VLMAX in all scenarios. In cases where the performance factor is less than 1, the performance is worse than the scalar C.

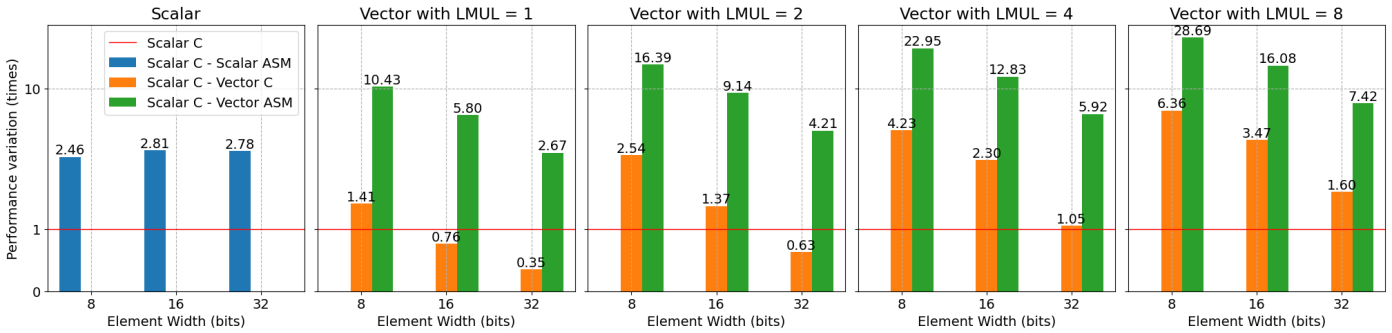

Fig. 3. Performance variation for different versions of the benchmark algorithm in comparison to the scalar C.

The scalar Assembly implementation shows an acceleration between 2.46 and 2.81 times faster than scalar C across different configurations due to the manual optimization in Assembly, simplifying code execution. The vector Assembly achieves the highest speedup among all configurations (10.43 to 28.69 times faster than scalar C) since it benefits from the parallelization and optimized assembly coding. The LMUL parameter significantly influences the performance since it affects the number of elements that a single instruction will operate. Also, smaller SEW values explore greater parallelism, resulting in higher performance.

The scenario of vector C, using intrinsic functions, shows performance variations ranging from 0.35 to 6.36 times compared to scalar C. The cases where the VEU performed worse than the scalar C occur because, unlike in the vector Assembly, where we execute the configuration instructions (setting the VL, LMUL, and SEW) only once at the beginning of the execution, the intrinsic functions run it every time before each operation, introducing repetitive overhead. Thus, vector C can reduce the performance compared to scalar C when the parallelization provided by vector operations is not enough to offset the repeated configuration overhead.

C. Fault Injection Simulation

We evaluated the output of the UART of each execution by using the Python script and the five categories of results mentioned in Section V-D. The experimental results are presented in Table II, which compares the results for different configurations of the HARV-SoC and HARV-SoC with the VEU. Thus, it presents the number of executions of each category achieved by each approach for the 1024 executions. Also, it presents the average number of fault injections, the total simulated fluence, the event cross-section, and the MWTF for each scenario.

The event cross-section gives an indication of the susceptibility of devices to radiation-induced faults since lower cross-section values mean lower vulnerability to faults. The results show an increase in the event cross-section values in the scenarios with the VEU since it increases the sensitive area. However, in the cases for the execution with the VEU and SEW = 8, the Mean Work to Failure (MTWF) was higher. This is because the executions with higher LMUL and smaller SEW values apply more parallelism and, therefore, have a shorter execution time. The difference in the execution time directly impacts the number of fault injections since the longer the execution lasts, the more faults are likely to be injected.

In the cases of SEW = 32, the amount of provided parallelism does not compensate for the increase in the sensitive area, resulting in fewer correct executions. In the hardened scenario, the difference in correct executions with and without the VEU is larger due to the application of hardening techniques, especially in the ALU and the register file, that are not applied for the vector ALUs and the VRF. Thus, even though fewer faults were injected, the MWTF is smaller.

Most of the benchmark errors in the scenarios with the VEU were caused by fault injections in the VRF. Unlike the main register file of the core, the VRF does not apply fault tolerance techniques due to the hardware overhead that would be required. Thus, the difference in the number of benchmark error executions between the vector and scalar executions is greater in the hardened scenario than in the baseline.

TABLE II
Fault injection executions

Execution classification	**Baseline**				**Hardened**			
	Scalar 8b	**Vector 8b**	**Scalar 32b**	**Vector 32b**	**Scalar 8b**	**Vector 8b**	**Scalar 32b**	**Vector 32b**
Correct	722	849	699	694	913	934	911	839
Benchmark error	32	27	54	101	15	31	15	84
Hang	7	2	5	1	1	0	0	0
Uncorrectable	198	109	196	145	89	49	92	60
Exception	65	37	70	83	6	10	6	41
Reliability analysis								
Average # of injections	18.70	11.11	18.73	19.36	21.98	13.63	21.52	19.44
Simulated fluence [n/cm^2]	$1.84 \cdot 10^{14}$	$8.92 \cdot 10^{13}$	$1.78 \cdot 10^{14}$	$1.49 \cdot 10^{14}$	$1.74 \cdot 10^{14}$	$9.20 \cdot 10^{13}$	$1.72 \cdot 10^{14}$	$1.31 \cdot 10^{14}$
Event XS [cm^2/device]	$1.61 \cdot 10^{-12}$	$1.94 \cdot 10^{-12}$	$1.80 \cdot 10^{-12}$	$2.22 \cdot 10^{-12}$	$6.31 \cdot 10^{-13}$	$9.79 \cdot 10^{-13}$	$6.59 \cdot 10^{-13}$	$1.41 \cdot 10^{-12}$
MWTF	3.47	5.92	3.20	3.11	9.31	11.38	9.06	5.54

D. Discussion

Although adding the VEU to the HARV-SoC introduces hardware, frequency, and power consumption overhead, it can significantly increase performance and reliability. The VEU can significantly accelerate computations, as for the vector Assembly application, where it achieved an execution up to 28.69 times faster than scalar C.

However, achieving these advantages depends on selecting the appropriate parameters, such as LMUL and SEW. Incorrect parameter choices can lead to degraded performance and reliability rather than improvement. For instance, in the hardened scenario with the VEU, using 8-bit elements increased the number of correct executions by 2.3%, but using 32-bit elements decreased it by 7.9% compared to the hardened scenario without the VEU.

Nevertheless, the VEU enhances reliability by reducing execution times and improving the MWTF metrics under the right conditions, i.e., with smaller elements. Thus, the VEU is a promising solution for applications that use multiplication operations and dependability is critical, particularly for 8-bit elements. This enhancement is particularly advantageous in image processing, where an 8-bit element represents a byte, especially in AI applications such as image classification.

VII. Conclusion

In this work, we have enhanced the VEU proposed in [8] by introducing support for vector multiplication and additional configuration instructions based on the Zve32x subset of the RVVE. The main goal was to accelerate the processing in applications requiring multiplication operations and assess its reliability. Also, we explored how different configurations of the VEU parameters can affect the reliability and performance.

We evaluated the impact of vector instructions over scalar ones to measure the processing time among different configurations of the VEU parameters. We also analyzed the impact of using applications in C and Assembly languages, in which we performed low-level optimizations. The results showed that the VEU proved very efficient, accelerating the execution up to 28.69 times when using the Assembly version of the benchmark. For the C version, the maximum acceleration was 6.36. However, we noticed that for VEU configurations where less parallelism is being applied (i.e., smaller LMULs with bigger SEWs), the use of the C version of the benchmark was not worth it since the parallelization provided by the VEU does not compensate for the repeated instruction overhead introduced by the compiler. The impact of this overhead is also present in the reliability evaluation, where for a SEW of 8 bits, the VEU enhances the fault tolerance, and for 32 bits, it worsens it.

In future work, we intend to conduct a comprehensive analysis of the impact on the reliability of all VEU parameters previously assessed for performance. This analysis will enable us to evaluate the benefits of utilizing C abstraction versus low-level optimization techniques to ensure reliability. Also, we aim to implement all the Zve32x extension instructions.

References

[1] RISC-V Organization, “RISC-V “V” Vector Extension,” https://github.com/riscv/riscv-v-spec, 2021, (Accessed on 05/22/2024).

[2] S. D. Mascio, A. Menicucci, E. Gill, G. Furano, and C. Monteleone, “Leveraging the openness and modularity of RISC-V in space,” *Journal of Aerospace Information Systems*, vol. 16, no. 11, pp. 454–472, 2019.

[3] G. Furano, A. Tavoularis, and M. Rovatti, “Ai in space: applications examples and challenges,” in *2020 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. IEEE, 2020, pp. 1–6.

[4] D. Cappellone, S. Di Mascio, G. Furano, A. Menicucci, and M. Ottavi, “On-board satellite telemetry forecasting with rnn on risc-v based multicore processor,” in *2020 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. IEEE, 2020, pp. 1–6.

[5] M. Yang, G. Hua, Y. Feng, and J. Gong, *Fault-tolerance techniques for spacecraft control computers*. Singapore: John Wiley & Sons, 2017.

[6] D. A. Santos, L. M. Luza, C. A. Zeferino, L. Dilillo, and D. R. Melo, “A low-cost fault-tolerant RISC-V processor for space systems,” in *2020 15th Design Technology of Integrated Systems in Nanoscale Era (DTIS)*, Marrakesh, 2020, pp. 1–5.

[7] D. A. Santos, A. M. P. Mattos, D. R. Melo, and L. Dilillo, “Enhancing fault awareness and reliability of a fault-tolerant RISC-V system-on-chip,” *Electronics*, vol. 12, no. 12, 2023.

[8] C. Imianosky, D. A. Santos, D. R. Melo, F. Viel, and L. Dilillo, “Implementation and reliability evaluation of a risc-v vector extension unit,” in *2023 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. IEEE, 2023, pp. 1–6.

[9] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[10] S. D. Mascio, A. Menicucci, E. Gill, G. Furano, and C. Monteleone, “On-board decision making in space with deep neural networks and RISC-V vector processors,” *Journal of Aerospace Information Systems*, vol. 18, no. 8, pp. 553–570, 2021.

[11] M. Johns and T. J. Kazmierski, “A minimal RISC-V vector processor for embedded systems,” in *2020 Forum for Specification and Design Languages (FDL)*. Kiel: IEEE, 2020, pp. 1–4.

[12] V. Maisto and A. Cilardo, “A pluggable vector unit for risc-v vector extension,” in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2022, pp. 1143–1148.

[13] S. Di Mascio, A. Menicucci, E. Gill, and C. Monteleone, “Extending the noel-v platform with a risc-v vector processor for space applications,” *Journal of Aerospace Information Systems*, pp. 1–10, 2023.

[14] M. Barbirotta, F. Minervini, C. R. Morales, A. Cristal, O. Unsal, and M. Olivieri, “Enhancing fault tolerance in high-performance computing: A real hardware case study on a risc-v vector processing unit,” *Authorea Preprints*, 2024.

[15] J.-G. Dumas, P. Lafourcade, J.-B. Orfila, and M. Puys, “Private Multi-party Matrix Multiplication and Trust Computations,” in *Proceedings of the 13th International Joint Conference on e-Business and Telecommunications (ICETE 2016) - Volume 4: SECRYPT*. SciTePress, 2016, pp. 61–72.

[16] D. Patterson and A. Waterman, *The RISC-V Reader: an open architecture Atlas*. San Francisco: Strawberry Canyon, 2017.

[17] A. Waterman, Y. Lee, D. Patterson, and K. Asanovic, “The RISC-V instruction set manual,” *Volume I: User-Level ISA', version*, vol. 2, 2014.

[18] RISC-V Organization, “Risc-v vector intrinsic document,” https://github.com/riscv-non-isa/rvv-intrinsic-doc, 2024, (Accessed on 06/27/2024).

[19] ——, “Gnu toolchain for risc-v,” https://github.com/riscv-collab/riscv-gnu-toolchain, 2024, (Accessed on 05/28/2024).

[20] J. L. Hennessy and D. A. Patterson, “Computer organization and design RISC-V edition: The hardware software interface,” 2017.

[21] G. A. Reis, J. Chang, N. Vachharajani, S. S. Mukherjee, R. Rangan, and D. I. August, “Design and evaluation of hybrid fault-detection systems,” in *32nd International Symposium on Computer Architecture (ISCA'05)*. IEEE, 2005, pp. 148–159.