Strada per strada : memoria e toponomastica politica a Torino

Algorithm-based Fault Tolerance for RISC-V Vector Processors in Safety-critical AI Applications

Sergiu-Mohamed Abed*†, Ahmet Cagri Bagbaba*, Connie O'Shea*, Josie E. Rodriguez Condia†, Matteo Sonza Reorda†
*Cadence Design Systems, Munich, Germany
†Department of Control and Computer Engineering, Politecnico di Torino, Turin, Italy
{*sergiu, abagbaba, oshea*}*@cadence.com* {*josie.rodriguez, matteo.sonzareorda*}*@polito.it*;

***Abstract*—Nowadays, edge AI is ubiquitous across many industries, including safety-critical domains where system failure can be catastrophic. Safety standards, such as IEC61508 and ISO26262, mandate that electronic systems must be able to detect faults during their operational life and act appropriately before a failure leads to unwanted consequences. Generally, fault detection is achieved through redundancy-based mechanisms, such as hardware- or software-level Dual Modular Redundancy (DMR), which have been proven highly effective, but at the cost of additional hardware area or significantly lower performance in the intended functionality.**

In this work, we propose a software-only safety mechanism specifically devised for Vector Processors. The approach is based on Algorithm-Based Fault Tolerance (ABFT) for matrix multiplication to achieve high fault-detection by leveraging checksums and the mathematical properties of matrix multiplication to detect dangerous faults affecting the final result of the operation. We implemented this mechanism by leveraging instructions from the RISC-V Vector Extension (RVV), and validated it on the Spatz Vector Cluster, considering stuck-at faults in the FPU addition/multiplication logic. Results show that the method achieves 97.53% dangerous fault detection and a 5 to 15 times speed-up wrt a baseline ABFT implementation.

***Index Terms*—Fault tolerance, Fault detection, Safety, Vector processor, RISC-V, Matrix multiplication**

I. Introduction

Nowadays, transistor miniaturization enables the integration of complex operations and functionalities into modern digital circuits, reducing power consumption and enabling the development of more advanced applications in the *Internet of Things* (IoT), edge, and High-Performance Computing domains.

In particular, the steadily increasing and sustained computational demands of certain critical applications, especially those using *Artificial Intelligence* (AI), are matched by the availability of specialized hardware accelerators, including domain-specific accelerators, and *vector processor* [1]–[3]. On the other hand, safety-critical domains, including robotics, healthcare, and autonomous/semi-autonomous systems, are characterized by stringent safety and reliability requirements (as mandated by standards such as IEC61508 and ISO26262) that call for the adoption of suitable fault-detection mechanisms. Such mechanisms are crucial, as modern semiconductor technologies are increasingly prone to faults arising in hardware during in-field operation [4]–[6].

This work was supported by the TIRAMISU project: EU Grant:101169378 HORIZON-MSCA-2023-DN-01

Fault detection mechanisms can be classified into three main groups: *i)* hardware mechanisms, *ii)* software-based mechanisms, and *iii)* hybrid strategies.

The first group consists of specialized structures that detect faults through redundancy, e.g., *Duplication with Comparison* (DWC), lock-step, Error Correcting Codes, or suitable test solutions, e.g., *Built-In Self-Test* (BIST). However, these mechanisms need to be introduced in the circuit during its development, and might entail significant silicon area costs and/or be subject to restrictions when used for in-field deployment. Finally, some of these mechanisms may not be available in Commercial off-the-shelf (COTS) components. Instead, the second group of fault detection strategies is based on algorithms and software-only routines. In particular, hardware-agnostic algorithms, such as *Algorithm-Based Fault-Tolerance* (ABFT), are effective for detecting faults affecting crucial application primitives, such as matrix multiplication [7]–[9], convolution [10], and FFT [11]. Indeed, the flexibility and scalability of the ABFT strategy, along with its ability to operate without hardware modifications, make it an attractive fault-detection solution for system integrators, even on COTS devices. The third group combines specialized hardware infrastructure with software commands to detect faults in a system. Unfortunately, such mechanisms require hardware changes during design, which is unfeasible in many systems (as previously discussed).

For configurable hardware accelerators, such as vector processors, flexible software-based approaches enable rapid and scalable deployment across varying numbers of parallel cores without introducing hardware overhead. Still, challenges arise regarding performance overhead, efficient use of available hardware resources, and fault-detection effectiveness.

In this work, we propose a Software-Based Fault-Detection mechanism for vector processors that leverages ABFT for matrix multiplication. In more detail, our approach uses vector instructions to efficiently compute checksums for the detection of permanent faults in the vector processor compute units.

To experimentally validate the proposed method, we used a reconfigurable RISC-V vector processor (Spatz) [12] comprising double-precision floating-point units (FPUs) and implemented the proposed method, resorting to the RISC-V Vector extension (RVV). The experimental results indicate that the proposed approach is effective at detecting a substantial percentage of permanent faults in the FPUs of the vector

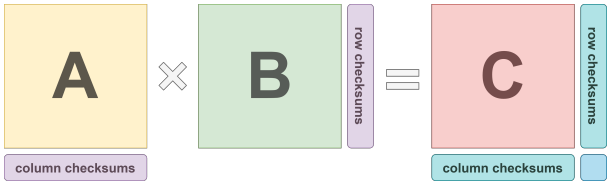


Fig. 1. Algorithm-based fault tolerance (ABFT): left matrix A extended with row of column checksums and right matrix B extended with column of row checksums. The result is a fully extended matrix C. The j^{th} column checksum of A is defined as $cc_j = \sum_{i=1}^{N} a_{ij}$. The i^{th} row checksum of B is $rc_i = \sum_{j=1}^{N} b_{ij}$.

processor. Moreover, the results show a performance boost of 5.10x-15.11x when comparing our vectorized approach to a baseline version not using the proposed techniques. Remarkably, the speed-up increases with the matrix size.

The key contributions of this work are the following:

1) Software-only, hardware-agnostic fault detection mechanism based on ABFT for matrix multiplication: this solution is applicable to any RVV-compliant vector processor.
2) Fast performance: our RVV-based ABFT achieves 5.10x-15.11x speed-ups compared to a baseline scalar version on Spatz.
3) Fault injection validation: large-scale permanent stuck-at fault injection campaigns targeting the FPU addition/multiplication logic show that the method detects about 97.53% of dangerous faults.

The remainder of the paper is structured as follows: Section II provides some background concepts. Section III describes the proposed ABFT-based strategy to detect permanent faults in vector processors. Section IV reports the experimental results gathered resorting to Spatz. Then, Section V concludes the paper and lists future works.

II. Background

A. Algorithm-based fault tolerance

Algorithm-based fault tolerance (ABFT) [7] is a well-known mechanism for fault detection and correction in matrix operations such as matrix multiplication, as illustrated in Fig. 1. The main idea of ABFT is to extend the matrices involved in the operation with checksums and to leverage the property of matrix multiplication stating that the result will also be a matrix extended with its own checksums. In detail, given the multiplication of two matrices A and B, ABFT works as follows:

1) compute a row vector of column checksums (i.e., sum of elements in a column) of *A* and append it to *A* as the last row;
2) compute a column vector of row checksums (i.e., sum of elements in a row) of *B* and append it to *B* as the last column;
3) perform $A \times B$ with the extended matrices to obtain an extended matrix C;
4) verify if the last row and last column of C correspond to the column and row checksums of C, respectively. If they differ, a fault has occurred and it was detected.

In principle, the ABFT strategy provides a hardware-agnostic mechanism to detect errors in matrix multiplication. Applying ABFT becomes more difficult when working with floating-point formats due to their inherent rounding errors. Because the checksums of C (calculated as described in steps 1 and 2) and its additional rows and columns (obtained through matrix multiplication) are computed through different sequences of operations, their difference will often not be equal to zero. Consequently, fault detection requires a predefined threshold to distinguish between errors due to floating-point rounding (below the threshold) and hardware faults (above the threshold).

Previous works focused on the adoption of ABFT for AI applications. In [13], the authors proposed an algorithm for the attention layer in transformer neural networks to include ABFT and implemented it in hardware. Their solution is effective: however, it is applicable for a very specific hardware platform and introduces some (albeit small) hardware overhead. In [14], the authors use a RISC-V instruction set simulator with fault injection support to inject bit-flip errors in the registers of a scalar processor to assess their ABFT implementation for scalar processors. Their approach can provide preliminary results, but the effectiveness evaluation of their solution at an abstraction level closer to the real hardware is missing. In this work, we propose an ABFT-based solution for RVV-compliant vector processors and assess its effectiveness wrt permanent faults.

B. Vector Processor Accelerators

Vector processors emerged in the 1970s as a way to accelerate massive scientific computations [15] and have recently reemerged to meet the increasing computational demands of edge AI. Furthemore, the recent approval of the RISC-V Vector instruction set extension (RVV) [16] paved the way for open-source vector processor designs, such as Spatz [12], Ara [17] and Vicuna [18].

Vector processors are hardware accelerators designed to exploit data-level parallelism by adopting the Single Instruction Multiple Data (SIMD) compute paradigm. Unlike scalar processors that must fetch and decode an instruction for each data element, a vector processor allows a single instruction to operate on an array of data elements. This reduces the time spent loading instructions from memory and decoding them and allows more efficient execution of mathematical computations. The arrays of elements are loaded into vector registers, which, unlike registers in scalar processors, are designed to hold multiple data elements. Due to their large capacity, vector registers significantly reduce the number of memory requests, allowing the loading of an entire array of data elements with a single load instruction. Generally, memory requests are slow, so reducing their frequency greatly improves performance.

III. Proposed method

This section describes a vectorized approach to efficiently implement ABFT in a multi-core vector processor. In this

work, we target architectures that implement the RISC-V instruction set architecture (ISA) with the RVV extension, but our solution can be applied to other vector ISAs as well.

Our ABFT-based strategy aims to detect permanent stuck-at faults in the FPUs of the vector processor. Although error correction could be easily added to the proposed approach, for the purpose of this paper, we only target the detection of permanent faults.

Any ABFT-based approach requires the efficient implementation of two phases: *checksum calculation* and *checksum verification*. In the following, we explain our strategies for their smart implementation.

Algorithm 1 Column partial checksum calculation by a core of the vector processor

Input: A, m_{start}, m_{end}, $coreID$
  $V1 \leftarrow [0, ..., 0]$
  for $i \in [m_{start}, m_{end}]$ **do**
    $V2 \leftarrow A[i, :]$ ▷ load row from memory
    $V1 \leftarrow V1 + V2$ ▷ accumulate
  end for
  $partial[coreID] \leftarrow V1$ ▷ store partial checksums in mem.

Algorithm 2 Summing partial checksums and appending as last row by core 0

Input: A, M, n_{cores}, $partial$
  $V1 \leftarrow [0, ..., 0]$
  for $coreID \in [1, n_{cores}]$ **do**
    $V2 \leftarrow partial[coreID]$ ▷ load from memory
    $V1 \leftarrow V1 + V2$ ▷ accumulate
  end for
  $A[M + 1, :] \leftarrow V1$ ▷ append as extra row in memory

Algorithm 3 Check on the extra row of result matrix C

Input: C, M, $threshold$
Output: $flag$
  $V1 \leftarrow$ $column\_checksums(C)$ ▷ as described in Algorithms 1 and 2
  $V2 \leftarrow C[M + 1, :]$ ▷ load $(M + 1)^{th}$ row from memory
  $V1 \leftarrow V1 - V2$
  $V1 \leftarrow abs(V1)$
  $max\_diff \leftarrow vredmax(V1)$ ▷ select highest value in $V1$
  if $max\_diff > threshold$ **then**
    $flag \leftarrow True$ ▷ fault detected
  else
    $flag \leftarrow False$ ▷ no fault

A. Checksum calculation

As shown in Algorithms 1 and 2 and illustrated in Fig. 2, checksums are computed in a vectorized fashion. The algorithms for computing row and column checksums are almost identical, differing only in whether columns or rows should be loaded in the vector registers $V1$ and $V2$ and in whether the result should be appended as the last row or column (see the last instruction in Algorithm 2). In the pseudo-code, we highlight the operations corresponding to vector instructions.

Considering a multicore vector processor, the matrix rows/columns are divided among the total number of cores n_{cores}, and each core computes the partial checksums on them (i.e., the intermediate results later used to calculate the true checksums). The partial checksums are calculated by iteratively loading a row/column into a vector register and summing them into an accumulator vector register. Once a core finishes summing the rows/columns assigned to it, the partial result is stored in memory.

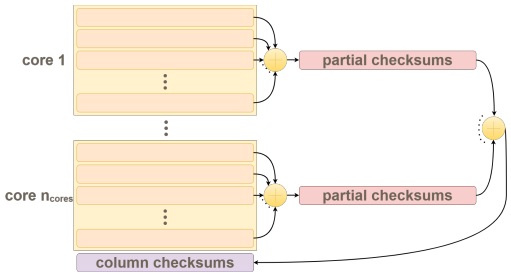
This approach ensures that we have as few instructions and memory accesses per row/column as possible. For a vector processor with n_{cores} cores and a matrix with N rows/columns, per core we have N/n_{cores} load instructions, N/n_{cores} addition instructions, and a single store instruction. Effectively, per row/column we have one load, one addition and n_{cores}/N store instructions.

Another, more straightforward approach of computing the checksums would be to simply load one row/column at a time into a vector register, perform a reduction instruction on it, and store the result in memory. However, this approach results in N/n_{cores} store instructions per core instead of just one as in our proposed method. An improvement to this method would be to slide the reduction results in another vector register and store the content of this register only at the end. The problem with this approach is that the number of instructions to be executed per row/column increases, making the benefits of reducing memory accesses unnoticeable.

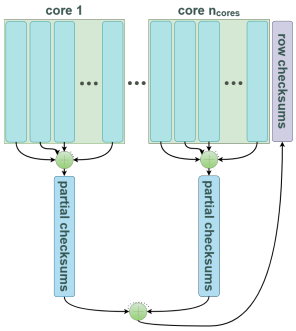
When all cores have finished, the partial results are further summed together, as described in Algorithm 2, and the result vector is appended as the last row/column to the matrix. We tried two approaches for summing the partial checksums: parallelized (across the n_{cores}) and on a single core. We noticed that the parallelized approach is slower than a single-core summation, due to the small vectors each core must sum. Generally, vectorization shows benefits for large vectors [19] [20]; therefore, splitting the partial checksums into short arrays, to be summed by each core, negatively impacts performance. In this case, the time it takes for a core to decide the subset of the partial checksums to operate on and to initiate the vector operation outweighs the benefits of parallelizing this task. For this reason, we chose the single-core approach.

B. Verification of checksums of matrix C

When multiplying the extended matrices A and B, we obtain a result matrix C extended with an additional row and column, which, in a fault-free scenario, correspond to the column and row checksums of C, respectively. According to Algorithm 3, the method first computes a vector of column/row checksums on C (resorting to Algorithms 1 and 2), then



(a) Column checksum calculation by vector summations of rows



(b) Row checksum calculation by vector summations of columns

Fig. 2. A general scheme of the vectorized checksum calculations for columns (a) and rows (b).

computes the difference between this vector and the extra row/column of C, and finally checks if the largest absolute value in the difference vector is larger or smaller than a certain threshold. If it is larger, then $flag$ indicates that a fault is detected. Otherwise, no fault is detected.

C. Threshold selection

The purpose of the threshold is to differentiate between the floating-point rounding error and the error introduced by a fault. Selecting a value for this threshold is challenging. If selected too high, a large number of faults causing errors below the threshold will remain undetected, whereas if selected too low, some rounding errors will be wrongly labeled as caused by faults (i.e., causing false positives).

To make sure we do not have false positives, when choosing the threshold we keep track of the sequence of operations performed for the calculation of max_diff in Algorithm 3. Then, we consider a "worst-case" scenario in which matrices A and B are initialized to the maximum value in the range of values $(-n, n)$ considered and calculate for each operation the maximum rounding error.

In the IEEE 754 [21] standard for floating-point arithmetic, the rounding error is bounded by $0.5ulp$, where ulp represents the distance between two consecutive floating-point numbers. With this bound, we calculate $0.5ulp$ for each operation result in the sequence of operations and accumulate it. After finishing the sequence, we use the accumulated error as the threshold.

IV. Experimental results

In this section, we first present the experimental setup. We then analyse the overhead introduced by the two ABFT versions and, finally, show their fault-detection effectiveness.

A. Set up

To experimentally assess the effectiveness of the proposed method, we focused on the Spatz Vector Cluster [12], which is a configurable RTL model of a multi-core RISC-V vector processor implementing the RVV extension. Its architecture (Fig. 3) consists of a L1 instruction cache and a banked scratchpad memory (L1 SPM) interconnecting a variable number of cores (processing elements, PEs). Each core consists of a variable number of FPUs and a vector register file (VRF) of $2KB$. The VRF consists of 32 vector registers (as defined by RVV

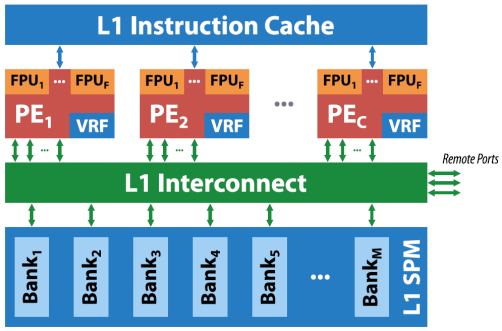


Fig. 3. Spatz Vector Custer architecture, adopted from [12].

extension), each of size $64B$, meaning that a single register can store up to 8 double-precision (64bits) elements. RVV also supports register grouping to form longer vector registers to hold longer arrays. We focused on two Spatz configurations, one with 2 cores ($Spatz_2$) and another with 4 cores ($Spatz_4$), both with L1 SPM of 16 banks and 4 FPUs per core.

We implemented an RVV-based version of the proposed method targeting Spatz, and compared it with a baseline version of ABFT. More in details, the baseline ABFT version is implemented in C language resorting to the classical nested loops approach and compiled to scalar instructions. Our RVV ABFT is implemented in C language with inline vector instructions, essentially removing the need for an inner loop and significantly reducing the number of memory accesses. Both baseline and RVV ABFT run bare metal across all cores.

We performed fault injections on the FPUs while running (30x30) matrix multiplication extended with ABFT, targeting the addition and multiplication (ADDMUL) blocks of the FPUs since these units dominate the computation in matrix multiplication and therefore represent the most safety-relevant hardware for the targeted workload. Each fault corresponds to a permanent stuck-at fault (SAF) on an internal signal (or register bit) within the selected FPU modules. For each injection run, the application output matrix C, the checksums and the ABFT decision flag are collected and compared against a fault-free golden run (same inputs and configuration) to study the effect of the faults.

To perform the experiments, we used the Cadence® Xcelium™ Fault Simulator (XFS) for fault simulation. We

TABLE I
RVV ABFT VS. BASELINE ABFT. PERFORMANCE CLOCK-CYCLES (CCS) AND SPEED-UP BY MOVING FROM BASELINE TO RVV ABFT

MatMul (NxN)	Spatz 2 cores			Spatz 4 cores		
	RVV ABFT (CC)	Baseline ABFT (CC)	speed-up	RVV ABFT (CC)	Baseline ABFT (CC)	speed-up
(30x30)	1,880	15,494	8.24	1,666	8,491	5.10
(32x32)	1,874	16,987	9.06	1,733	8,855	5.11
(34x34)	2,056	19,378	9.43	1,776	10,392	5.85
(36x36)	2,100	20,701	9.86	1,832	10,964	5.98
(38x38)	2,268	23,128	10.20	1,876	12,535	6.68
(40x40)	2,290	25,990	11.35	1,971	13,186	6.69
(42x42)	2,475	28,557	11.54	2,007	14,947	7.45
(50x50)	2,921	38,222	13.09	2,291	20,458	8.93
(60x60)	3,587	54,206	15.11	2,718	27,769	10.22

wrote various Python scripts to automate configuration setup (e.g., changing Spatz configurations, matrix sizes, and value ranges) and to analyze the effects of faults at the end of each fault injection. It is worth noting that the approach proposed in this paper also applies to other tool flows.

To evaluate the effectiveness of the ABFT fault detection mechanism, each injected stuck-at fault is classified according to its impact on the application-level results and the ABFT detection response. Fault injection outcomes are categorized in the following ways:

- *Application-Specific Safe* [22]: faults not affecting the result of matrix multiplication;
- *Dangerous Detected*: faults causing wrong results that are detected by ABFT;
- *Dangerous Undetected*: faults causing wrong results that are not detected by ABFT.

Since we are working with floating-point numbers, mismatches in the checksums can be either due to floating-point rounding error or due to faults. To differentiate between the two cases, we further divide *Dangerous Undetected* in two subcategories: *Dangerous Undetected below threshold*, where the errors due to faults fall below the threshold (as explained in subsection III-C), and *Dangerous Undetected above threshold*, in which errors are larger than the threshold.

B. RVV-based ABFT performance with respect to Baseline ABFT

Table I compares the time required by the baseline and RVV-based ABFT implementations and the speed-up introduced by the latter on different Spatz configurations. The speed-up ranges from 8.2 to 15.11 and from 5.10 to 10.22 on $Spatz_2$ and $Spatz_4$, respectively. Our results indicate that *the larger the matrix dimensions, the higher the speed-up* for both vector processor configurations, since increasing the size of the matrices leads to fewer instructions to be fetched and decoded per data element, and to fewer memory accesses per data element in the case of RVV-based ABFT, while for the baseline the number of such operations increases with the matrix dimensions.

It can be noticed that the speed-up figures on $Spatz_4$ are not as high as on $Spatz_2$. This is caused by bank conflicts in the L1 SPM, where multiple data element accesses refer to the same bank. Doubling the number of cores does not always result in double performance, as multiple cores could stall waiting for the same bank to be available.

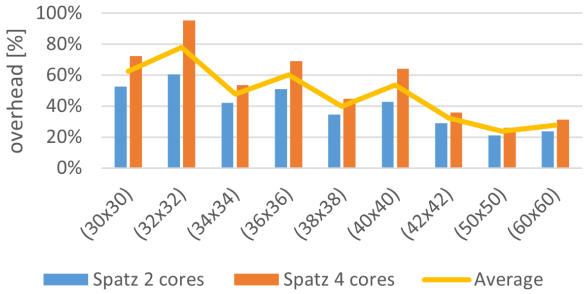


Fig. 4. Performance overhead introduced by RVV-based ABFT with respect to non-hardened matrix multiplication across different matrix dimensions. (NxN) refers to multiplication of two matrices of dimension (NxN).

In baseline ABFT, when moving from 2 to 4 cores, we see the number of clock cycles (CCs) decreasing by 45.20% to 49.27%. On RVV-based ABFT, when doubling the number of cores, we do not see a corresponding drop in CCs. However, the percentage CC drop increases when the size of the matrices increases (between 11.38% to 24.23%). This is because, when the size of the matrices increases, so does the number of vector summations performed by each core. The reason why RVV-based ABFT does not have a decrease in CCs similar to the baseline is because there are parts of the source code that cannot be vectorized (e.g., deciding on which part of the matrix a core should work on). For small matrices, the overhead from the non-vectorizable operations outweighs the benefits of the few vector instructions. The benefits of vectorization come with larger matrices.

C. RVV-based ABFT performance with respect to non-hardened matrix multiplication

Fig. 4 summarizes the performance overhead (i.e., the percentage of extra CCs) introduced by RVV-based ABFT with respect to the non-hardened matrix multiplication across different matrix dimensions for the two considered Spatz configurations. We observe that the overhead ranges from 21.18% to 60.49% and from 26.02% to 95.19% on $Spatz_2$ and $Spatz_4$, respectively. The overall trend is that *the larger the matrices are, the lower the performance overhead is*. This is due to the quadratic increase in the dimension of the result matrix C, while the number of instructions performed by ABFT increases only linearly. Moreover, the number of store instructions during the partial checksum calculations remains constant, equal to the number of cores n_{cores}.

When looking at Fig. 4 we can notice a pattern in which two consecutive matrix dimensions cause an increase in overhead but when moving to the third dimensions the overhead decreases, moving below the previous two. For example, when moving from (30x30) to (32x32), the overhead increases, but for (34x34) the overhead falls below the previous two cases. This is because the matrix-multiplication algorithm performs loop unrolling to reuse data in the register file to compute two rows of C in the same loop iteration. For example, considering matrices of dimension (30x30) and $Spatz_2$ performing matrix

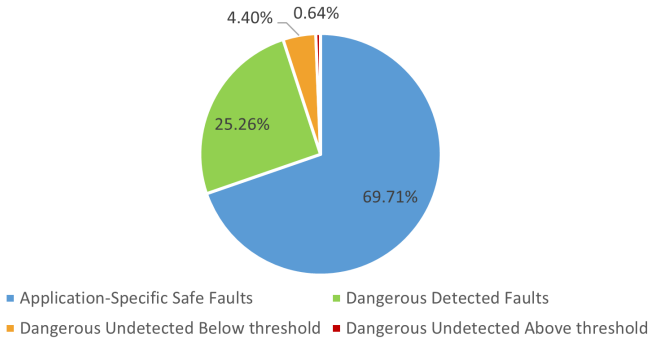


Fig. 5. Fault injections outcomes.

multiplication, each core processes 15 rows of C and each core does loop unrolling to process two rows during the same loop iteration. If we consider the extra row added by ABFT, one core will process 15 rows of C while the other will process 16 (for a total of 31). Since two rows are processed per loop iteration, the core with 15 rows will have to run the last iteration for just one row (being 15 an odd number), whereas the other core with 16 rows will run the same number of iterations, but the last one processing two rows without having to load significantly more data from memory to the registers. So, the extra row does not lead to significantly more overhead in this case.

For matrices of dimensions (32x32), one core will process 16 rows, while the other will processe $16 + 1$ due to ABFT. Since 16 is an even number, this means the core with 16 rows runs 8 loop iterations, while the other will have one extra iteration where it needs to load data from memory.

The same trend is noticed for $Spatz_4$. If C has 30 rows, two cores will process 8 rows and the other two 7 rows each. By adding an extra row to C, three cores process 8 rows each and the last core 7. Since the cores run in parallel, the overall overhead is not so large (we need to keep in mind that C also includes an extra column, which makes the rows longer by one element). If C has 32 rows, then all cores process 8 rows each. By having an extra row, one core will process 9 rows, triggering an extra loop iteration to process the last row and leading to more load and store operations.

Fig. 4 shows that $Spatz_4$ imposes a larger overhead on matrix multiplication than $Spatz_2$. As mentioned in subsection IV-B, this is caused by L1 SPM bank conflicts, which are more likely to occur with a larger number of cores accessing the memory. This effect is not so strong on matrix multiplication because of the usage of loop unrolling to reuse the data available in the register file multiple times.

The memory overhead introduced by RVV ABFT (stemming from the additional rows and columns on the extended matrices and from storing the partial checksums in memory) ranges between 3.34% and 6.70%, and between 4.45% and 8.93% on $Spatz_2$ and $Spatz_4$, respectively. The memory overhead decreases as the matrix dimensions increase.

D. Fault detection capabilities

We assessed the fault detection effectiveness of RVV-ABFT on $Spatz_2$ and $Spatz_4$ while performing (30x30) matrix multiplication with values in ranges (-1, 1) and (-100, 100). This results in 4 different setups combining the Spatz configuration and the range of values, and we performed two fault-injection campaigns per setup, resulting in a total of 8 campaigns. During each campaign, we injected 100% of the stuck-at faults at the RT level in the ADDMUL module of the FPUs, amounting to 41,520 and 83,040 faults for $Spatz_2$ and $Spatz_4$, respectively.

Fig. 5 summarizes the results of the experiments. All fault injection campaigns produce almost the same distribution of faults among the different categories, varying only by $\pm 0.69\%$, with Dangerous Undetected above threshold varying by $\pm 0.09\%$. What can be noticed is that a significant amount of SAFs ($\sim$ 69.71%) are application specific safe faults. This is because the ADDMUL unit consists of multiple lanes with different precisions. Specifically, it consists of 8 lanes: 1 lane 64 bits wide, 1 lane 32 bits wide, 2 lanes 16 bits wide and 4 lanes 8 bits wide. Since our analysis resorts to double-precision (64 bits) matrix multiplication, nearly all faults present in the other lanes will not affect the operation.

The percentage of faults detected by our RVV-ABFT solution is 25.26%. When excluding the application-specific safe faults (as they do not have any effect on the application), we achieve a percentage of detected dangerous faults of $\sim$ 83.37%. If we further consider dangerous faults below threshold as safe (as their effect is similar to FP rounding errors), we achieve a dangerous fault detection of $\sim 97.53\%$.

The few faults labeled as Dangerous Undetected above threshold are caused by faults affecting a signal belonging to a handshake interface controlling the flow of data into and out of the FPU, present in each lane of ADDMUL. This specific signal tells that the data on the corresponding interface is valid and stable. We noticed that a stuck-at-1 in this signal in all lanes will affect the application. On the other side, a stuck-at-0 on this signal impacts only the 64bit lane.

V. Conclusions

This work introduced a software-based fault detection mechanism for permanent stuck-at faults in the FPU ADDMUL logic of RISC-V vector processors using the ABFT strategy. We implemented this mechanism leveraging vector instructions from the RVV ISA extension to achieve high fault detection while keeping performance overhead as low as possible.

We analyzed the effectiveness of our RVV ABFT solution on the Spatz Vector Cluster and we compared its performance to a baseline ABFT implemented with scalar instructions, assessed the performance overhead introduced on top of matrix multiplication and evaluated its fault detection capability.

The experimental results show that our RVV ABFT solution achieves 5.10x-15.11x speed-ups compared to the baseline version, reaches performance overheads as low as 21.18% and achieves dangerous fault detection of 97.53%.

Future work will more deeply assess the effectiveness of our RVV ABFT strategy when considering its impact in the case where the Vector Processor executes a whole Neural Network.

REFERENCES

[1] B. Dally, "Hardware for deep learning," in *2023 IEEE Hot Chips 35 Symposium (HCS)*. IEEE Computer Society, 2023, pp. 1–58.

[2] L. Kljucaric *et al.*, "Architectural analysis of deep learning on edge accelerators," in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, 2020, pp. 1–7.

[3] M. Meribout *et al.*, "State of art iot and edge embedded systems for real-time machine vision applications," *IEEE Access*, vol. 10, pp. 58 287–58 301, 2022.

[4] C. Constantinescu, "Trends and challenges in vlsi circuit reliability," *IEEE Micro*, vol. 23, no. 4, pp. 14–19, 2003.

[5] S. Hamdioui *et al.*, "Reliability challenges of real-time systems in forthcoming technology nodes," in *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2013, pp. 129–134.

[6] S. Majji *et al.*, "A study on the comprehensive analysis of electro migration for the nano technology trends," in *2020 6th International Conference on Advanced Computing and Communication Systems (ICACCS)*, 2020, pp. 898–901.

[7] K.-H. Huang *et al.*, "Algorithm-based fault tolerance for matrix operations," *IEEE Transactions on Computers*, vol. C-33, no. 6, pp. 518–528, 1984.

[8] F. T. Luk *et al.*, "An analysis of algorithm-based fault tolerance techniques," *Journal of Parallel and Distributed Computing*, vol. 5, no. 2, pp. 172–184, 1988.

[9] S. Bal *et al.*, "A novel fault-tolerant architecture for tiled matrix multiplication," in *2023 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2023, pp. 1–6.

[10] K. Zhao *et al.*, "Ft-cnn: Algorithm-based fault tolerance for convolutional neural networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 7, pp. 1677–1689, 2021.

[11] L. L. Pilla *et al.*, "Software-based hardening strategies for neutron sensitive fft algorithms on gpus," *IEEE Transactions on Nuclear Science*, vol. 61, no. 4, 2014.

[12] M. Cavalcante *et al.*, "Spatz: Clustering compact risc-v-based vector units to maximize computing efficiency," Sep. 2023.

[13] V. Titopoulos *et al.*, "Custom algorithm-based fault tolerance for attention layers in transformers," in *2025 IEEE 38th International System-on-Chip Conference (SOCC)*. IEEE, 2025.

[14] U. Sharif *et al.*, "Efficient software-implemented HW fault tolerance for TinyML inference in safety-critical applications," in *2023 Design, Automation Test in Europe Conference Exhibition (DATE)*. IEEE, 2023, pp. 1–6.

[15] R. M. Russell, "The CRAY-1 computer system," *Communications of the ACM*, vol. 21, no. 1, pp. 63–72, 1978.

[16] K. Asanovi'c *et al.*, *RISC-V "V" Vector Extension Specification, Version 1.0*, RISC-V International, 2021, ratified standard.

[17] M. Perotti *et al.*, "Ara2: Exploring single- and multi-core vector processing with an efficient rvv 1.0 compliant open-source processor," *IEEE Transactions on Computers*, vol. 73, no. 7, pp. 1822–1836, 2024.

[18] M. Platzer *et al.*, "Vicuna: A Timing-Predictable RISC-V Vector Coprocessor for Scalable Parallel Computation," in *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), B. B. Brandenburg, Ed., vol. 196. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, pp. 1:1–1:18. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2021/13932

[19] H. Shan *et al.*, "Performance characteristics of the cray x1 and their implications for application performance tuning," in *Proceedings of the 18th Annual International Conference on Supercomputing (ICS)*. ACM, 2004, pp. 175–183.

[20] M. Perotti *et al.*, "A "new ara" for vector computing: An open source highly efficient RISC-V V 1.0 vector processor design," in *2022 IEEE 33rd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2022, pp. 1–9.

[21] IEEE, "IEEE Standard for Floating-Point Arithmetic," *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84, 2019.

[22] A. C. Bagbaba *et al.*, "Automated identification of application-dependent safe faults in automotive systems-on-a-chips," *Electronics*, vol. 11, no. 3, 2022. [Online]. Available: https://www.mdpi.com/2079-9292/11/3/319