FAULTLESS: Flexible and Transparent Fault Protection for Superscalar RISC-V Processors

Moritz Waser[1](✉), David Schrammel[2⋆], Robert Schilling[2], and Stefan Mangard[1]

[1] Graz University of Technology
firstname.lastname@tugraz.at
[2] Rivos Inc.
{davidschrammel,rschilling}@rivosinc.com

Abstract. Fault injection (FI) attacks pose a significant threat to the reliability and security of devices. They can cause data or control-flow corruption, leading to system failure or allowing malicious attackers to steal secret data or leak cryptographic keys. To protect against faults, many vendors extend their processors with lockstep capabilities, which require either dedicated hardware duplication or a reconfigurable second core that can act as a shadow core. The former causes a large hardware overhead while the latter requires an inflexible configuration during boot time with additional implications for software design. Software-based fault protection requires recompilation of existing code with custom compilers, which introduces compatibility issues.
This paper presents FAULTLESS: A fault protection mechanism that transparently performs hardware-based instruction duplication and utilizes the existing redundancy in superscalar processors. Contrary to lockstep approaches, our design facilitates a flexible protection approach with marginal hardware overhead that allows developers to toggle the fault protection during runtime, providing a choice between security and performance. The design is fully transparent and compatible with preexisting binaries. We implement our prototype based on the *VeeR EH1* RISC-V processor and show that, when active, our fault protection generates an average performance overhead between 32 % and 79 %, depending on the hardware configuration. Non-critical applications can deactivate the feature and run without any overheads. On an Artix-7 FPGA, our hardware modifications incur a minimal overhead of 3.5 % for LUTs and 2.8 % for flip-flops.

Keywords: Fault-Protection, RISC-V, Superscalar CPU

1 Introduction

Faults can have tremendous effects on both the reliability as well as the security of a system. For example, a fault of natural origin, e.g., cosmic radiation, may lead to

⋆ The work was done while the author was at Graz University of Technology.

system failure of an embedded device, causing monetary loss or, even worse, harm to humans. Apart from naturally occurring faults, malicious attackers can perform precise fault injection attacks using power or clock glitches, electromagnetic interference or even lasers. This enables them to escalate privileges [35], bypass security measures [38], or break the security guarantees of ciphers [4]. The traditional threat model for fault injection attacks only considers adversaries with physical access to a device. However, recent research [11,21,33] has shown that faults can even be injected remotely.

Fault protection measures require redundancy, which can be realized in software, hardware, or both [32]. Software-based redundancy, such as instruction duplication, provides hardware portability at the cost of program recompilation and large performance overheads [2,3,6]. As it lacks any consideration for the underlying microarchitecture, it also fails to protect against precise fault injection [18]. Redundancy in hardware can circumvent the performance overhead by increasing the required silicon area and power. A full lockstep design, while providing strong protection, will require more than twice the die area and power [29]. Commodity systems, such as Infineon's TriCore [14] or NXP's S32K3 family [23], give developers the option to either run their system with a multicore configuration or utilize additional cores as secondary checker cores in a lockstep approach. While this provides a tradeoff between additional security and performance, the configuration is set during boot time and cannot be altered at run time. In addition, the duplicated hardware is limited to the CPU. Peripherals like memory require additional fault protection, e.g., through Error Correction Codes (ECCs) such as Hamming codes [13]. Mixed designs achieve a high level of protection with balanced overheads but require an inflexible hardware-software co-design [40]. As embedded devices are limited by area, power, and performance requirements, designers must strike a balance between effective and performant fault protection and implementation costs.

Contribution

In this paper, we introduce Faultless, a fault-protected system design approach for superscalar RISC-V processors that aims for flexibility in terms of performance and application, as well as low hardware overhead. The core idea of the design is a hardware-based, flexible repurposing of a superscalar core's additional execution pipelines, which provide complimentary redundancy as a fault protection measure. Faultless ensures that no instruction can commit a corrupted result to the microarchitectural state while providing a simple, interrupt-based recovery mechanism. The design emphasizes flexibility, as the fault protection can be toggled during runtime, allowing developers to protect important code sections at the cost of a performance overhead while retaining full performance otherwise. This focus stems from the observation that many fault injection attacks in the past specifically targeted security-critical computations like password or signature checks [38]. While large performance overheads are not tolerable in many use cases, dynamically protecting devices from faults during critical operations is an attractive prospect.

We implement and evaluate a prototype of our FAULTLESS design based on the *VeeR EH1* [7] RISC-V core. Furthermore, our security analysis highlights the extensive fault protection of the design. When the fault protection is active, the evaluation shows a performance overhead between 33 % and 79 %, depending on the hardware configuration. We compare the logic element usage on an Artix-7 FPGA for the unmodified *VeeR EH1* core and our prototype and find that our design uses only 3.5 % more LUTs and 2.76 % more flip-flops. Finally, we open-source our prototype [3] to facilitate future research.

In summary, our main contributions are as follows:

1. We present FAULTLESS, a system design for superscalar processors that facilitates fault protection with small hardware overhead and flexible performance impact.
2. We detail how our design utilizes existing redundancy to reduce the hardware overhead compared to traditional lockstep designs.
3. We present a proof-of-concept prototype based on the *VeeR EH1* RISC-V core and highlight required hardware changes.
4. We evaluate our prototype to showcase the small hardware overhead of 3.5 % for LUTs and 2.76 % for flip-flops, as well as the variable performance overhead between 0 % and 79 %.
5. We extensively analyze the security of our design, underlining all advantages of our design compared to lockstep designs.

2 Background

This section presents fundamental background knowledge on faults, fault attacks and common fault protection measures, which is required for subsequent chapters.

2.1 Fault Attacks

In a fault attack, the adversary induces a glitch into the circuit of a chip and exploits the various effects on the physical level, e.g., transient voltage fluctuations or timing violations [28]. While naturally occurring faults, e.g., caused by exposure to cosmic rays, are mostly of interest for data centers [16] and systems that operate in harsh environments [19], targeted fault attacks are actively used to break the security of embedded devices [24,36]. Exploiting this allows attackers to bypass security measures [35,38], alter the control-flow [11], or leak cryptographic secrets [4,8].

A fault is categorized by its spatial and temporal properties, such as time, duration and location of the fault, as well as its effect. The effect of a fault attack can be described on multiple abstraction layers. On the physical layer, the fault may charge the gate of a transistor, which leads to a bit-flip in the CMOS logic. For the register transfer layer, this manifests as a computation error. Within the running program, this error can lead to an erroneous comparison, e.g., a wrong branch during a password authentication.

[3] `https://extgit.isec.tugraz.at/sesys/faultless`

Classic fault attacks require physical access to a device, which allows an attacker to induce electromagnetic pulses, clock glitches, power glitches, or even shoot a laser directly at the decapsulated silicon die [15]. However, works like *Rowhammer.js* [11], *Plundervolt* [21], or *CLKSCREW* [33] presented remote fault attacks mounted solely through software, which further increased the attack surface of fault attacks.

2.2 Fault Protection

Protecting circuits from faults always requires a form of redundancy, which can be realized in software and hardware. The primary goal of fault protection is to detect the occurrence of a fault. Secondary goals are the correction of the fault's effects and pinpointing the fault location.

We distinguish between three common types of redundancy: information, spatial and temporal [32]. Information redundancy, e.g., parity bits, entails adding redundant information to stored or transmitted data. Spatial redundancy describes the physical replication of data or hardware. Temporal redundancy is achieved when a circuit performs a computation multiple times in sequence.

Implementing redundancy in software is attractive because the solution is microarchitecture-agnostic and thus easily applicable to a range of devices. However, software fault protection incurs large performance overheads [2,3,6]. Common software countermeasures include code duplication [34], range checks [27], and signature-based control-flow-integrity schemes [22].

Unlike software countermeasures, redundancy in hardware is tailored for a specific microarchitecture. Hardware fault protection comprises ECCs, e.g., Hamming codes [13], shadow registers [26], and modular redundancy [20]. In addition to redundant hardware, designers can also include measures to shield the device from the cause of a fault or detect physical effects like clock variations or charged particle impacts [37,41].

3 Threat Model

Independent of whether a fault is caused by natural effects or by a malicious attacker, our design goal is to protect processors from single bit-flips in registers and logic lines. We consider protection functional when we can successfully detect any single bit-flip within the processor. We choose a single bit-flip model to showcase the feasibility of our countermeasure. The presented prototype does not protect against multiple bit-flips, but the proposed design facilitates scaling to higher degrees of protection.

Potential adversaries may inject faults anywhere within a CPU, with any methodology, *i.e.*, physical access to the device or remotely. The goals of attackers include crashing systems, or hijacking the control-flow. Both can be achieved either directly through a corrupted Program Counter (PC) or branch instruction, altered register values or control signals, or by corrupting data that subsequent instructions depend on. Although our design partially protects systems from

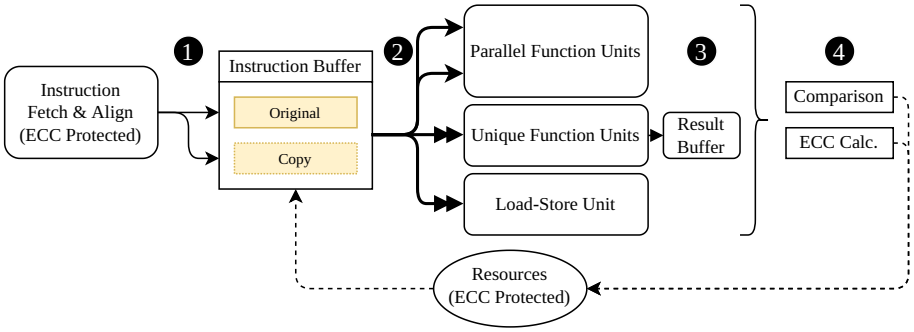

Fig. 1. Concept overview of the FAULTLESS design.

the effects of permanent, *i.e.*, *stuck-at* faults, we do not generally claim it does. Denial-of-Service (DoS) attacks that aim to delay or block further execution are out of the scope for this work.

4 Design

This section introduces the fundamental concept of the FAULTLESS design and highlights the design rationale. We first present a high-level overview of the concept and then discuss the implications for specific system components in depth.

4.1 Overview

To achieve extensive fault protection for a given system, *i.e.*, a processor and its peripherals, it is paramount to protect both the individual components as well as the communication between them. FAULTLESS is a full system design that achieves fault protection through a combination of hardware-based instruction duplication and ECCs. Unlike lockstep designs, which duplicate the entire processor, we protect the processor pipeline itself through redundant execution of instructions that keeps the hardware overhead small by exploiting the existing redundancy in a superscalar pipeline. Interactions with peripherals are protected with purposefully placed ECCs. In addition, fault protection is controlled through Control and Status Registers (CSRs), which can be modified by software at any given time. With this, developers have full control over the level of protection and related performance overheads, which allows them to selectively protect critical code sections while causing no overhead for other code.

Figure 1 highlights the main contribution of this work, which lies in the hardware-based instruction duplication. The design focuses on the execution path between the issue stage, where instructions leave the instruction buffer, and the commit stage, where results are evaluated and the architectural state is updated. Our design consists of four distinct changes to a superscalar pipeline. First (❶),

we duplicate all instructions as they are added to an instruction buffer. Second (❷), we issue both instances of an instruction to available function units. The issuing strategy depends on the availability and presence of redundant function units. Double-headed arrows represent sequential issuing and, thus, temporal redundancy. Third (❸), for instructions that can only be issued sequentially, we introduce an intermediate result buffer when necessary. Finally (❹), we compare the results of both instances.

To complement the protected processor, we assume that all peripherals are protected by ECC or stronger means of protection.

Other processor components, like branch prediction logic, are of little concern to fault protection, as any corrupted state will be eventually corrected by other parts of the microarchitecture. In the case of branch prediction, a corrupted prediction will always be compared to the result of the actual computation of the branch condition at a later point. The point of duplication depends on the microarchitecture, but it must ensure continuous protection. When instructions are protected by ECC as they are fetched, the duplication must happen simultaneously with the ECC decoding and possible error correction. As they pass through the pipeline, duplicated instructions are protected by either spatial or temporal redundancy, depending on instruction type and availability of function units, e.g., Arithmetic Logic Units (ALUs). To keep the hardware overhead small, we only introduce comparisons of the results and their destinations at the final commit point. Whenever an instruction pair, *i.e.*, the original instruction and its redundant copy, can be issued in parallel, the result can be compared at the final pipeline stage without additional buffering. Instruction pairs that were issued sequentially may require an additional result buffer, which holds the result and target of the first instance until the second instance completes execution. In Out-of-Order (OoO) architectures, this buffering can be optimized through the already existing reorder buffer. To ensure full redundancy, the forwarding paths of all instruction instances must be separated. In case of duplication, one instance is marked with a special copy flag that limits available forwarding paths, such that original instructions can only forward results to other original instructions and copies can only forward to copies, respectively. When the design is scaled up to multiple copies, instruction copies must be assigned an identifier that enables distinction of multiple forwarding paths. The same mechanism also prevents instructions from forwarding results to their own copies when source and destination registers are identical.

4.2 Peripherals

Next to the processor pipeline, all interactions with peripherals and the peripherals themselves, *i.e.*, memory, must be protected to achieve full fault protection. This includes main memory, data and instruction caches, and all additional buffers required for e.g., memory transactions. We assume a system where all peripherals are protected by ECCs, meaning all faults that target the peripherals directly are covered.

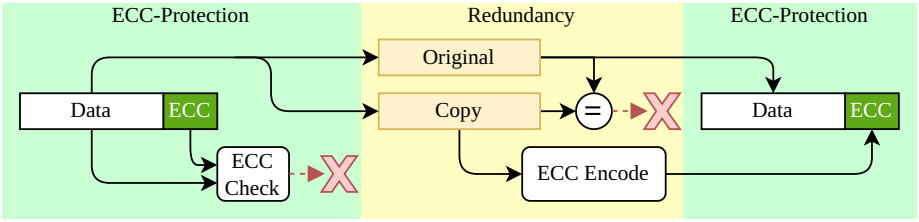

Fig. 2. The transition between ECC-protected domains and the redundancy-protected processor pipeline. The ECC check and the duplication of data happen simultaneously in combinatorial logic. The red cross symbolizes a raised exception due to an ECC check or a failed comparison between duplicated data.

Implementations must ensure that handshakes between peripherals and the processor provide a seamless transition between ECCs and other forms of redundancy. This is exemplified in Figure 2. When receiving data from a peripheral, the ECC check and the duplication of the value must happen in combinatorial logic within the same clock cycle. The receiving registers and the ECC decoder must be connected to the same source, such that any fault on the data line is directly detected before the value is latched. Similarly, when sending data to a peripheral, the redundant data must be compared in parallel to the ECC encoding.

4.3 Register Instructions

We consider all instructions that do not interact with the memory interface as register instructions. This includes arithmetics, branches, CSR manipulation, fences, and trap-return operations. For all instructions that are computed by a function unit with several instances, we exploit the existing spatial redundancy. Instructions pairs are issued in parallel and reach their commit point simultaneously, allowing for simple comparison. When instructions require a function unit that only exists once, e.g., a multiplier, we issue them in succession. As we cannot let individual instances of duplicated instructions commit their result, e.g., to the register file, we may require an additional buffer in the commit stage that stores the result of the first instruction for an additional cycle. The requirement depends on the availability of the result within the pipeline and the forwarding capabilities of the given microarchitecture. When the results of all instances of an instruction are available within the pipeline when the first instance reaches the commit point, the buffer is not necessary as the results can be directly compared. OoO processors can optimize this through the reorder buffer. When an additional buffer is necessary, it must be extended with forwarding capabilities, similar to an additional pipeline stage.

For performance optimizations, a pipeline may also include additional stalling points between the issue and commit stages. This complicates forwarding, as instructions that already have passed such a stalling point may have left the pipeline when the stalling condition is resolved. If the microarchitecture solves

this by introducing additional buffers for intermediate results, the pipeline must enforce similar forwarding constraints on these buffers as for normal pipeline stages.

4.4 Memory Instructions

To protect loads and stores to storage devices or other memory-mapped peripherals, we must ensure that the values and target addresses are always protected by a form of redundancy for the full transaction. Moreover, duplication of memory instructions requires special attention as they might exhibit non-idempotent properties, *i.e.*, they cause side effects when issued twice. For this reason, we classify all memory-mapped peripherals into two groups: internal and external.

Internal devices include all peripherals that reside on the same SoC and have known behavior in terms of side effects. This encompasses both devices that exhibit strictly idempotent behavior, such as flash or SRAM, as well as non-idempotent behavior, e.g., an interrupt controller.

External peripherals are all devices that are connected to the SoC through a bus. Since external devices are generally unknown from a processor designer's perspective, we assume all such devices as behaving non-idempotent.

For internal peripherals without side effects, we achieve fault protection through instruction duplication, similar to register instructions. Both loads and stores are duplicated and issued sequentially. As described in Section 4.2, the microarchitecture must ensure full redundancy for the entirety of the transaction. In this regard, memory interfaces require special assessment, as they often include performance optimizations such as instruction merging for subsequent transactions to the same address.

For internal peripherals with side effects, as well as external peripherals, we require a handshake that transforms the spatial/temporal redundancy within the processor to other forms of protection, e.g., ECCs. The same mechanism as described in Section 4.2 can be used to achieve this. As long as the memory instructions are within the processor, we protect them in the same fashion as all other instructions. The issue step, the address computation, and the propagation down the pipeline until the operands are written to a bus buffer or similar, are protected through redundancy. From this point onwards, other forms of redundancy (e.g., ECC) must ensure the integrity of the data. In the case of loads, we require the inverse of the described handshake to protect all values from the moment they enter the processor pipeline. With this, we ensure that all values are always protected by redundancy while avoiding potential issues related to non-idempotent behavior.

4.5 Mode Transition and Recovery

FAULTLESS enables flexible fault protection that can be toggled through software. To control the mode of operation, we add two CSRs, which can be modified by a regular CSR-write instruction.

First, the `u_protectionmode` CSR controls if the fault protection using instruction duplication is active or not, *i.e.*, whether instructions get executed redundantly and results are compared. The correct mode of operation must be enforced for all instructions that are fetched after the CSR-write instruction. For a seamless transition between modes, writing to `u_protectionmode` in the processor's commit stage triggers a full pipeline flush. When enabling fault protection, this prevents all in-flight instructions following the CSR-write from committing their results without being protected by redundancy. After the pipeline flush, the instructions following the CSR-write are re-fetched and executed redundantly. Because the `u_protectionmode` CSR controls the behavior of the entire pipeline, including result comparisons and forwarding logic, the pipeline must also be flushed when exiting fault-protected mode. This could optionally also be achieved by instrumenting a compiler such that it always places a fence after writes to `u_protectionmode`.

Second, the `u_detectionmode` CSR controls how a detected fault within the processor pipeline should be handled. To deal with detected faults, FAULTLESS implements full forward-error-correction. Faults within the pipeline are detected as soon as the affected instruction reaches its commit point. When `u_detectionmode` is set, a detected fault raises an exception that can be handled in software, which enables further diagnosis. If the CSR is not set, a detected fault triggers a full pipeline flush and rolls back execution to the oldest in-flight instruction. With this, the processor retries executing the same instruction sequence that contained the fault, without requiring any software handling. A detected fault has the highest priority and should override any other microarchitectural effects.

5 Implementation

This section provides details about the prototype implementation of FAULTLESS, which is based on the *VeeR EH1* (formerly known as *SweRV EH1*) RISC-V core that features a 32-bit, superscalar, in-order, dual-issue, 9-stage pipeline with a single privilege level and no virtual memory. The core can be configured to include Tightly Coupled Memory (TCM) that is protected with ECC for both instructions and data.

5.1 Overview

The pipeline consists of three instruction-fetch and align stages, a decoding stage, and five execute stages. A general overview of the application of our FAULTLESS design to the core is given in Figure 3. Bold arrows symbolize the issue step, which is handled differently, depending on the instruction type and the availability of function units. Specifically, ALU instructions (including branches) can be issued in parallel, while all other instructions are issued sequentially. All arrows pointing in two directions describe memory interactions, either with TCM or with external devices. All other arrows indicate the path of instructions and their corresponding results within the pipeline. Until instructions reach the decode

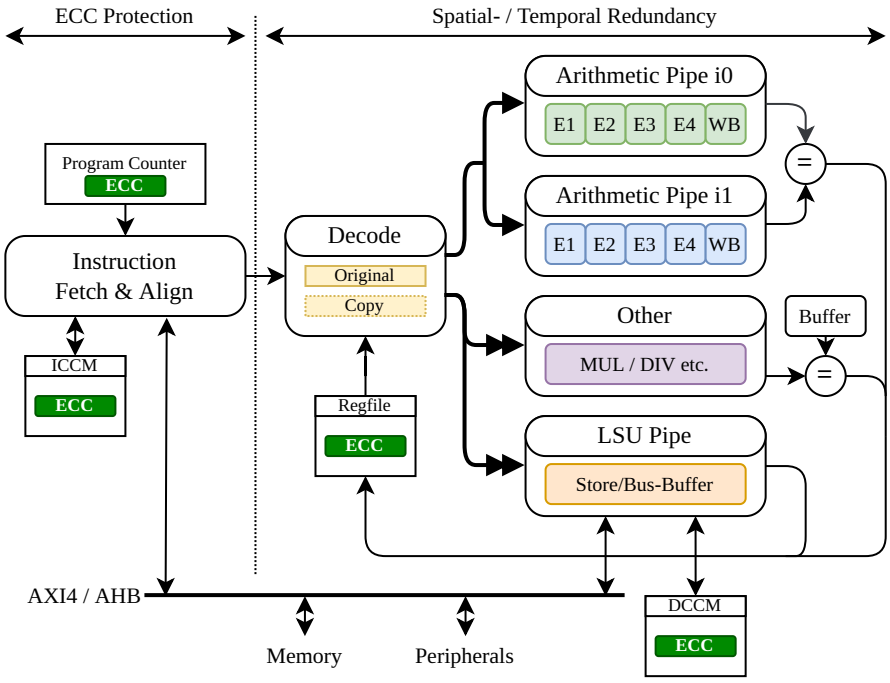

Fig. 3. General overview of our FAULTLESS concept applied to the *VeeR EH1* core.

stage, they are protected by ECC. From the decode stage onward, the duplication provides redundancy to protect the execution.

The decode stage contains an instruction buffer holding up to four instructions, of which two may be issued at every cycle. When ECC-protected instructions reach the decode stage, we ensure continuous protection following Section 4.2. The execute stages contain two ALU pipelines, a multiplier, a division unit, and a Load Store Unit (LSU). ALU operations can be either computed immediately or delayed for three cycles for complex forwarding between pipelines.

The following sections highlight our changes to the *VeeR EH1* core by tracing the path of instructions from their duplication to the final commit. Our changes must ensure that instruction pairs are issued together, flow down the pipeline redundantly and commit their result after a comparison confirms that no fault has occurred.

5.2 Issue Step

An instruction buffer within the decode stage is responsible for managing the way individual instructions are issued. All instructions that are written to this buffer are duplicated when fault protection is enabled. The buffer must ensure that all duplicated instructions are issued together, following their designated

protection strategy. We modify the issuing behavior such that arithmetics (except multiplication and division), branches, fences, CSR and system instructions will be issued in parallel and are, thus, protected by spatial redundancy. All other instructions are issued sequentially, protecting them with temporal redundancy.

The buffer is implemented as a queue that opportunistically issues the two oldest instructions while receiving new instructions from the align stage. To implement FAULTLESS, we extend the buffer's functionality in two distinct ways. First, we implement instruction duplication by writing incoming instructions to the buffer twice. When doing this, the second instruction is marked as a copy. Second, we modify the issuing behavior such that we ensure that instruction pairs, *i.e.*, the original and its copy, are always issued following the correct redundancy strategy. This is necessary because the buffer could otherwise issue instructions that should be protected with spatial redundancy in parallel to instructions that are issued sequentially. Whenever possible, two different but sequentially issued instructions will be parallelized.

5.3 Execution Pipeline

When instruction pairs are issued, either in parallel or sequentially, they normally receive their operands either from the register file or from subsequent pipeline stages. To improve performance by avoiding stalling conditions, the *VeeR EH1* core can allow instructions to flow down the pipeline with unresolved dependencies that are resolved at later stages. Independent of whether this is the case, we limit the forwarding logic to avoid two newly introduced hazards, as described in Section 4.1.

First, instructions must never forward results to their own copy. This would lead to wrong behavior when the destination of a copied instruction is equal to an operand. In the *VeeR EH1* core's pipeline, we stop this from happening for both parallel and sequential instruction pairs.

Second, there must always be two forwarding events for every pair of instructions. One instance of an instruction can never be allowed to forward its result to a pair of subsequent instructions, as this would undermine the redundancy. In the case of the *VeeR EH1* core, this is only a problem because the core has several commit points, which are not all at the final execute stage. If the microarchitecture ensures that all instructions commit their result at the same point, the second condition for the forwarding logic can be dropped, as a fault will always be detected before the corrupted value that was forwarded has any effect.

Memory operations are handled by the LSU, which is responsible for calculating target addresses and delegating requests to their corresponding buffers. The *VeeR EH1* core distinguishes memory accesses between internal targets, e.g., TCM, and external, bus-bound peripherals. We issue all loads and stores redundantly to protect the entire address calculation and the corresponding data until we know whether the address is internal or external. Accesses to TCM are resolved instantly, which allows for these instructions to be resolved within the 5 execute stages without stalling. Since TCM access behaves idempotently, we

can protect these instructions with full redundancy, similar to other instructions. When dealing with external targets, the memory request is passed to a bus controller after three stages. We ensure continuous fault protection by implementing a handshake that validates the integrity of the request while it is passed to the bus controller alongside a newly calculated ECC. For loads, the pipeline is stalled at this point until the load is resolved. As soon as the result is available, we read it from the bus controller twice while checking the ECC to again ensure continuous protection.

The overall stalling behavior of the core also requires special attention. In certain conditions, the core can stall only the first three execute stages while the fourth and fifth stages continue execution. To resolve dependencies between the stalled and active stages, after the stalling condition is resolved, the core buffers the results of the final two stages. Since these buffers may also be affected by a fault, we add comparators that verify the integrity of these buffers whenever they hold valid data.

5.4 Commit Points

The *VeeR EH1* core has several commit points that must be extended with comparison logic. All instructions besides branches and bus-bound memory transactions commit their result at the final execute stage. While branches may commit at the final stage, they can also already report a mispredicted branch at the third execute stage, causing a partial pipeline flush. For this reason, the results of both the final and the third stage must be compared for every parallelized instruction pair. Instructions that are issued sequentially also commit at the final stage but require an additional buffer to store the result of the first instance. When the second instance reaches the commit point, the result is compared to the buffer and written to the register file. We also extend the result buffer with forwarding logic to avoid the case where other instructions can only receive a forwarded value from the second instance. In parallel with every result comparison, we calculate a new ECC that is stored alongside the result.

5.5 CSRs and Register File

Following Section 4.5, we add two CSRs to the core: `u_protectionmode` controls the currently active protection mode (either normal or FAULTLESS operation) and `u_detectionmode` determines, whether a detected fault triggers an exception or a pipeline flush with subsequent re-fetching of the faulted instruction sequence. Writing to `u_protectionmode` triggers a full pipeline flush to ensure the protection of all following instructions. Since the *VeeR EH1* core by default only includes logic to process CSR instructions in one arithmetic pipeline, we extend the second pipeline with the same capabilities so that we can issue CSR operations in parallel. In addition, we add ECC protection for the PC, all CSRs and all general-purpose registers. This requires encoders and decoders for all related read and write ports.

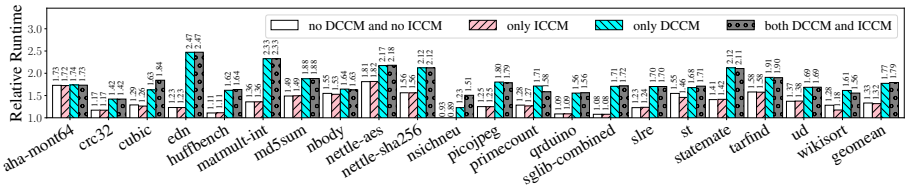

Fig. 4. Relative runtime overhead and geometric mean for all memory configurations for all Embench benchmarks.

6 Evaluation and Analysis

In the following, we evaluate the performance and hardware overheads of our design using the Embench benchmark suite [10] and FPGA synthesis. In addition, we comprehensively analyze the fault protection of the FAULTLESS design and our prototype.

6.1 Performance

We evaluate the overall performance overhead of FAULTLESS by comparing the runtime (cycles) of the Embench benchmark suite for the base *VeeR EH1* core and our modified design. Our performance evaluation is based on hardware simulation using *Verilator* [39]. As the *VeeR EH1* core is envisioned as an embedded device that runs bare-metal software, we run all benchmarks in this fashion. We found that repeated runs of individual benchmarks show only negligible runtime differences, rendering warmup runs irrelevant. Because multiple runs of the same benchmark provide neither additional information nor follow a normal distribution, we do not report standard deviation. The runtime of any program on the *VeeR EH1* core depends heavily on the configuration regarding TCM usage, both for data (DCCM) and code (ICCM). For this reason, we evaluate performance for all four combinations of TCM usage. When we use the TCM, we preload both code and data of every benchmark into it before starting the simulation. In addition, we also place the top of the stack within the tightly coupled data memory.

Figure 4 shows the individual testbench results for all configurations, as well as the corresponding geometric mean. The biggest overhead of 79 % occurs when both data and code are placed within the TCM. Not using the TCM at all or using it only for code leads to an overhead of 32 %. The reason why faster data access results in a significant overhead increase is twofold: First, TCM access is instant and does not cause pipeline stalling. Contrary to this, accessing regular memory frequently causes pipeline stalls that mask the additional overhead introduced by our instruction duplication. Second, as TCM access is resolved instantly and requests are sent sequentially, it always causes 100 % overhead for memory instructions. In comparison, memory transactions with the bus controller only cause a stall once, as the actual memory access is not performed twice. A

noteworthy effect occurred in the *nsichneu* testbench, which performed better when the fault protection was enabled in a configuration without DCCM. When executed without protection, this testbench causes frequent pipeline stalls due to dependencies on loads. The added latency of the instruction duplication allows the pipeline to optimize this, which overall improves the performance.

6.2 Hardware Overhead

The design goal of Faultless is to provide flexible fault protection with negligible hardware overhead. To evaluate this overhead, we synthesize both the original and our modified *VeeR EH1* design for a *Nexys A7* FPGA board and compare the utilization of logic elements. Our modifications do not change the critical path of the core, as the added logic is computed in parallel to much longer combinational paths. The parameters (TCM size, etc.) are identical, so all differences stem from the core itself. Table 1 shows the overhead for both LUTs and flip-flops. Our modifications increase the number of LUTs by 3.5 % and the number of flip-flops by 2.76 %, underlining the benefit compared to traditional lockstep designs. The largest overhead is located in the decoding and execute stages, which contain the instruction buffer, the register file, and the forwarding logic.

6.3 Security Analysis

The fault protection of our Faultless design is achieved through a combination of different measures: hardware-based instruction duplication for the processor pipeline, ECCs for vulnerable processor state (e.g., general purpose registers or CSRs) and peripherals, and protected communication between components.

Fetched instructions, as well as their related addresses, should always be protected by ECC until they reach the issuing or decoding stages. The PC must also be protected through ECC or duplication and comparison. As long as instruction integrity is guaranteed, branch predictors require no modifications, as faults in predictions may cause delays but can never corrupt the system state. Any corrupted branch prediction will be corrected by the actual target computation, while mispredictions that were corrupted into correct predictions will be detected by the redundant execution. When instructions are duplicated as they are placed into a buffer, the ECC decoding must occur in parallel to

Table 1. FPGA utilization (LUTs and flip-flops) and relative overhead for all modules of the base *VeeR EH1* and our Faultless design.

	LUTs	Flip-Flops	Combined
Baseline	28978	12293	41271
Faultless	29991	12632	42623
Overhead	3.5 %	2.76 %	3.28 %

guarantee continuous redundancy. The protection through duplicated instructions is centered around the comparison that happens at each commit point. This comparison must be as restrictive as possible. Only identical instruction pairs may pass the comparison, which also protects the system from all faults that occur within pipeline registers for control signals. Instructions are considered identical if they are valid, their results and destination match, and one instruction is marked as a copy. Following our threat model, the comparison logic itself does not require additional protection. A bit-flip may occur in either one instruction instance or the comparison logic itself. In the former case, the comparison will detect the fault, while in the latter case, the detection logic may be triggered without any violation of instruction integrity. Both cases lead to a detected fault that can be corrected by reissuing the affected instruction sequence. A corrupted instruction can never pass the comparison logic, as the fault can never corrupt the instructions and the comparison at the same time. The additional result buffer that may be needed for instructions protected through temporal redundancy is also protected by the restrictive comparison. As long as a fault does not turn a valid instruction into an invalid instruction, which would immediately raise an exception, corrupted instructions will not be detected immediately but as soon as they reach their commit point. Propagation of corrupted results within the pipeline is prohibited by the restrictive forwarding logic for original and copied instructions, which have completely separated data paths. When all instructions commit their result at the same point, restricting the forwarding logic is not required since a corrupted instruction will always be detected before a following instruction can commit a corrupted result.

System state such as the PC, the register file, or CSRs must be protected from faults alongside the processor's pipeline. Otherwise, a single bit-flip could corrupt data-dependent control flow or alter microarchitectural behavior, e.g., disable instruction duplication by modifying the `u_protectionmode` CSR. The PC and register file can be protected with ECCs that are checked at every access. System registers that passively control the processor's functionality require special attention, as checking an ECC upon reading them is not sufficient to prevent faults from silently corrupting system behavior. Thus, critical CSRs, such as our `u_protectionmode` must be protected with a continuous self-check. This can be achieved using combinatorial ECC checks, which also allow correction. Alternatively, shadow registers that fully duplicate the system's state can be used. This offers detection capabilities that exceed regular ECCs at the cost of higher hardware overhead and, assuming just a single copy of the state, no error correction.

For loads and stores, the degree of fault protection depends on additional measures implemented by the target. Side-effect-free stores to memory-mapped peripherals, like the TCM for the *VeeR EH1* core, can be protected with full redundancy in the entire pipeline. The integrity of loads from such devices depends on the integrity of the data itself, which can be ensured by the use of, e.g., ECC. Implementations must ensure continuous redundancy, as explained in Section 4.4. Our instruction duplication protects both the address calculation

and the data itself. Bus-bound targets of memory operations that may exhibit side effects require additional protection of the bus protocol and related buffers. Our FAULTLESS design only guarantees correct address calculation and, for stores, data integrity until the data is added to such a buffer. Whether the integrity of a loaded value from such a device is maintained depends on the device. From the moment the value enters the processor pipeline, it is again protected by duplication.

Our design is limited with regard to the amount and type of faults that occur in a system. When multiple transient faults occur simultaneously, the proposed measure cannot guarantee full protection. Similarly, when fault effects are permanent, it depends on the location of the fault whether our design can still function correctly. A single *stuck-at* fault can be corrected when it occurs in a register with ECC, but it can cause permanent erroneous behavior in other places.

7 Related and Future Work

Existing work on fault protection can be roughly categorized into software- and hardware-based designs. The performance and hardware overheads of the discussed designs are collected in Table 2.

The idea of idle execution unit utilization for fault tolerance in software was first mentioned by Schuette et al. [31]. They integrate a software control-flow monitor into idle slots of VLIW instructions without additional hardware modifications. Franklin [9] discusses possible fault targets in processors and mentions redundant execution based on idle function units as a defensive measure. Oh et al.[25] partition general purpose registers into two groups and duplicate instructions to operate on both groups individually. They compare the register sets after each basic block to detect errors. HAFT [17] presents a similar scheme by duplicating data flows and relying on the superscalar pipeline to optimize the execution of two independent threads. Chen et al. [5] utilize SIMD instructions on commodity hardware to duplicate data flow. All of these schemes require custom compilers and exhibit worst-case performance overheads above 100 %.

Table 2. Performance and hardware overheads of related works compared to FAULTLESS.

		Performance [%]	Hardware [%]
Software-based	HAFT [17]	100	0
	EDDI [25]	13-111	0
	SIMD [5]	364	0
Hardware-based	Lockstep [1]	0	≥100
	[30]	0	75
	SHAKTI-V [12]	25	20
	FAULTLESS (This work)	0-79	3.28

In comparison, FAULTLESS is fully compatible with legacy binaries and offers the possibility to flexibly toggle the fault protection and reduce the introduced overhead to zero.

Lockstep designs [1,14,23] achieve comprehensive fault coverage by duplicating a system at a specific granularity, e.g., at package or processor level. This approach causes none or only negligible performance overhead, but requires at least a hardware overhead of 100 %. Fault-protected RISC-V designs [12,30] achieve their protection through a combination of ECC and modular redundancy. While this provides reasonable protection, it also causes a significant hardware overhead. Compared to this, our design keeps the hardware overhead minimal while presenting a flexible trade-off between protection and performance.

Potential future work could implement and evaluate our FAULTLESS design on a larger, out-of-order core. Such a core provides higher flexibility and could better optimize dependencies, leading to a smaller performance overhead. The performance implications of adding a data cache to the memory hierarchy could also be examined. Scaling up the duplication to triplication and studying the potential for latency-free fault correction is also possible. This would also provide a possibility to detect permanent faults in specific function units, which could be disabled to ensure correct operation.

8 Conclusion

In this paper, we presented FAULTLESS, a hardware design approach for superscalar processors, which provides fault protection at minimal hardware overhead and dynamic performance overhead. The design performs generic instruction duplication purely in hardware, making it compatible with preexisting binaries. Application developers or the OS can toggle the fault protection, including the performance overhead, during runtime, allowing them to add targeted protection to critical code sections. With this, we present an alternative to lockstep designs that protect processors with a brute-force approach that causes high hardware overheads. Our design achieves fault protection while providing a methodology that can easily be scaled for stronger threat models and larger processors. We implemented and evaluated a proof-of-concept prototype of the design based on the *VeeR EH1* core, showing that extensive fault protection can be achieved with minimal hardware overhead for superscalar processors. Furthermore, we provide a security analysis of our design, highlighting possible fault targets and protective measures used to detect state corruption.

9 Acknowledgements

We thank the anonymous reviewers for their valuable feedback that improved this work. This project has received funding from the Austrian Research Promotion Agency (FFG) via the AWARE project (FFG grant number 891092). Additional funding was provided by generous gifts from Intel.

References

1. Baleani, et al.: Fault-tolerant platforms for automotive safety-critical applications. In: CASES'03 (2003)
2. Barenghi, et al.: Countermeasures against fault attacks on software implemented AES: effectiveness and cost. In: WESS'10 (2010)
3. Barry, et al.: Compilation of a Countermeasure Against Instruction-Skip Fault Attacks. In: CS2@HiPEAC'91 (2016)
4. Boneh, et al.: On the Importance of Checking Cryptographic Protocols for Faults (Extended Abstract). In: EUROCRYPT'97 (1997)
5. Chen, et al.: SIMD-based soft error detection. In: CF'16 (2016)
6. Cojocar, et al.: Instruction Duplication: Leaky and Not Too Fault-Tolerant! In: CARDIS'17 (2017)
7. Digital, W.: RISC-V: high performance embedded SweRV core microarchitecture, performance and CHIPS Alliance. `https://riscv.org/wp-content/uploads/2019/04/RISC-V_SweRV_Roadshow-.pdf` (2019), accessed: 2023-08-18
8. Dobraunig, et al.: SIFA: Exploiting Ineffective Fault Inductions on Symmetric Cryptography. IACR Trans. Cryptogr. Hardw. Embed. Syst. (2018)
9. Franklin, M.: Incorporating fault tolerance in superscalar processors. In: HIPC'96 (1996)
10. Free, Foundation, O.S.S.: Embench: Open Benchmarks for Embedded Platforms. `https://github.com/embench/embench-iot/` (nd), accessed: 2023-01-13
11. Gruss, et al.: Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In: DIMVA'16 (2016)
12. Gupta, et al.: SHAKTI-F: A Fault Tolerant Microprocessor Architecture. In: ATS'15 (2015)
13. Hamming, R.W.: Error Detecting and Error Correcting Codes. The Bell System Technical Journal (1950)
14. Infineon: 32-bit AURIX TriCore Microcontroller. `https://www.infineon.com/cms/de/product/microcontroller/32-bit-tricore-microcontroller/` (nd), accessed: 2024-04-15
15. Karaklajic, et al.: Hardware Designer's Guide to Fault Attacks. IEEE Trans. Very Large Scale Integr. Syst. (2013)
16. Keller, A.M., Wirthlin, M.J.: The Impact of Terrestrial Radiation on FPGAs in Data Centers. ACM Trans. Reconfigurable Technol. Syst. (2022)
17. Kuvaiskii, et al.: HAFT: hardware-assisted fault tolerance. In: EUROSYS'16 (2016)
18. Laurent, et al.: Cross-layer analysis of software fault models and countermeasures against hardware fault attacks in a RISC-V processor. Microprocess. Microsystems (2019)
19. Luza, et al.: Impact of Atmospheric and Space Radiation on Sensitive Electronic Devices. In: ETS'22 (2022)
20. Lyons, R.E., Vanderkulk, W.: The Use of Triple-Modular Redundancy to Improve Computer Reliability. IBM J. Res. Dev. (1962)
21. Murdock, et al.: Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In: S&P'20 (2020)
22. Nicolescu, et al.: Software Detection Mechanisms Providing Full Coverage Against Single Bit-Flip Faults. IEEE Transactions on Nuclear Science (2004)
23. NXP: S32K3 Microcontrollers for Automotive General Purpose. `https://www.nxp.com/products/processors-and-microcontrollers/s32-automotive-platform/s32k-auto-general-purpose-mcus/`

s32k3-microcontrollers-for-automotive-general-purpose:S32K3 (nd), accessed: 2024-04-15
24. O'Flynn, C.: BAM BAM!! On Reliability of EMFI for in-situ Automotive ECU Attacks. IACR Cryptol. ePrint Arch. (2020)
25. Oh, et al.: Error detection by duplicated instructions in super-scalar processors. IEEE Trans. Reliab. (2002)
26. Pattabiraman, et al.: Dynamic Derivation of Application-Specific Error Detectors and their Implementation in Hardware. In: EDCC'06 (2006)
27. Rela, et al.: Experimental Evaluation of the Fail-Silent Behaviour in Programs with Consistency Checks. In: FTCS'96 (1996)
28. Richter-Brockmann, et al.: Revisiting Fault Adversary Models - Hardware Faults in Theory and Practice. IEEE Trans. Computers (2023)
29. Rodrigues, et al.: Towards a Heterogeneous Fault-Tolerance Architecture based on Arm and RISC-V Processors. In: IECON'19 (2019)
30. Santos, et al.: Characterization of a RISC-V System-on-Chip under Neutron Radiation. In: DTIS'21 (2021)
31. Schuette, M.A., Shen, J.P.: Exploiting Instruction-Level Resource Parallelism for Transparent, Integrated Control-Flow Monitoring. In: FTCS'91 (1991)
32. Sorin, D.J.: Fault Tolerant Computer Architecture (2009)
33. Tang, et al.: CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management. In: USENIX Security'17 (2017)
34. Theißing, et al.: Comprehensive analysis of software countermeasures against fault attacks. In: DATE'13 (2013)
35. Timmers, et al.: Controlling PC on ARM Using Fault Injection. In: FDTC'16 (2016)
36. Timmers, N., Mune, C.: Escalating Privileges in Linux Using Voltage Fault Injection. In: FDTC'17 (2017)
37. Upasani, et al.: Framework for economical error recovery in embedded cores. In: IOLTS'14 (2014)
38. Vasselle, et al.: Laser-Induced Fault Injection on Smartphone Bypassing the Secure Boot-Extended Version. IEEE Trans. Computers (2020)
39. Veripool: Verilator. https://www.veripool.org/verilator/ (nd), accessed: 2024-04-18
40. Werner, et al.: Protecting the Control Flow of Embedded Processors against Fault Attacks. In: CARDIS'15 (2015)
41. Yuce, et al.: FAME: Fault-attack Aware Microprocessor Extensions for Hardware Fault Detection and Software Fault Response. In: HASP'16 (2016)