A Fast Low-Level Error Detection Technique

Zhengyang He†, Hui Xu‡, Guanpeng Li†
† University of Iowa, Iowa City, IA, USA
‡ Fudan University, Shanghai, China
zhengyang-he@uiowa.edu, xuh@fudan.edu.cn, guanpeng-li@uiowa.edu

***Abstract*—As transistors continue to shrink in size, the soft error rate in computer systems is rising, posing a critical threat of severe failures. error detection by duplicating instruction (EDDI) has been proposed as a prominent software-based technique for detecting soft errors. However, utilizing EDDI specifically at the assembly level has not been well explored in the literature. Towards this end, the paper introduces FERRUM, an innovative assembly level EDDI, which is a boosted version compared with the original assembly level EDDI by using SIMD and other compiler-level transformations. We evaluate FERRUM in both fault coverage and runtime performance compared with IR level EDDI and original assembly level EDDI. The results show that FERRUM not only ensures 100% protection coverage at the assembly level but also surpasses baseline techniques by more than 50% in runtime performance overhead.**

***Keywords*—Silent Data Corruption, Error Resilience, Fault Injection, Instruction Duplication**

I. INTRODUCTION

Transient hardware faults, commonly known as soft errors, have been increasingly prevalent due to ongoing trends in hardware design, including the reduction in the sizes of transistors and operating voltages [1], [2]. These faults, often triggered by atmospheric particles such as alpha or neutron particles, pose a significant risk of silent data corruptions (SDCs) which potentially lead to incorrect program outputs, seriously compromising the dependability of modern computer systems [3]–[5]. Traditional hardware-based protection strategies, such as voltage guard bands, hardware redundancy, and circuit hardening methods, though effective in mitigating these faults, have become less viable due to their substantial overheads in performance and energy consumption [6], [7].

To overcome the challenges, researchers have proposed software solutions [5], [8]–[11]. To date, error detection by duplicating instructions, or EDDI, is a particularly popular technique that has been applied in many application scenarios [5], [8], [12]. EDDI duplicates instructions at compile time and detects mismatch at runtime if any of the two copies is corrupted due to errors. The technique requires no modifications to hardware and thereby provides a more flexible and often less resource-intensive alternative compared with traditional hardware-based solutions.

Instruction duplication can be deployed across various system levels. However, a vast number of existing methods are implemented at LLVM intermediate representation (IR) level [5], [8], [12]–[16], whereas it is rare that the technique is engineered from lower layer such as assembly level. The reasons are threefold: First, LLVM IR benefits from a well-established set of openly available compiler tools and libraries supported by many communities from both academia and industry to facilitate the implementation [17], [18]. Secondly, the IR excels in conducting detailed program analysis, offering the advantage of application-specific designs. Finally, there has been a substantial body of existing research and open library on instruction duplication, supplying the basic building blocks for implementing the technique [8], [13], [14]. As a result, implementing EDDI at assembly level remains largely underexplored.

In the past, there have been studies that show existing EDDI techniques in the literature, that is, those implemented at IR level, suffer from non-negligible loss of error coverage in the protection [13], [19]. The technique often shorts on the error detection coverage of SDC even though a program is fully protected by duplicating all the IR instructions. The issues are acute especially when the protected programs are evaluated with a more realistic fault injection technique such as those conducted at assembly level, not to mention using more representative methods such as beam testing [20]. As a result, there is a strong demand to explore EDDI at lower layer of software stack where is close to the occurrence of hardware faults

In this paper, we propose a method that implements, optimizes and evaluates EDDI at assembly level, which will support research and engineering studies in the area. We observe that there is possible potential to improve EDDI coverage and performance at assembly level via exploring the under-utilization in CPUs with x86 ISA as well as compiler-level optimizations. We carefully engineer the technique, FERRUM, that duplicates and protects assembly instructions of a program. Our evaluation shows that FERRUM provides 100% SDC coverage at assembly level in contrast to 72% provided by existing IR-level EDDI, with nearly 52% speed-up in runtime performance overhead. *To our best knowledge, we are the first ones who dive into the optimization of the assembly-level EDDI at the lower layer, conducting an end-to-end evaluation and analysis of the protections with existing state-of-the-arts for both performance and fault coverage.*

Our main findings are as follows:

- We evaluate existing IR-level EDDI technique and observe there is a non-negligible gap (28% on average) between the anticipated SDC coverage and the actual measured one when evaluating at a lower layer such as assembly level.

2158-3927/24/$31.00 ©2024 IEEE
DOI 10.1109/DSN58291.2024.00023

- By studying the working principle of EDDI, we replicate an implementation of existing EDDI at assembly level. In our evaluation, we observe that the SDC coverage reaches 100% when evaluating at assembly level. The observation confirms that the protection at lower layer can be much more effective in terms of fault coverage due to the absence of uncertainty from the backend cross-layer compilation process.
- However, the performance overhead of assembly-level EDDI is higher compared with existing IR-level EDDI (by 30% on average). We analyze the root causes and find that the source of the additional overheads is mainly from the additional unprotected footprint generated by the backend compiler when compiling from IR to assembly.
- We further explore the optimization opportunities at assembly level when designing EDDI. We find that it is possible to leverage under-utilized resources such as SIMD capability of modern processors to optimally engineer EDDI at assembly level in order to improve the performance and space overhead.
- Based on the findings, we deliver our technique, FERRUM. Our evaluation shows that FERRUM achieves an average of 52% speed-up in runtime performance while maintaining full SDC coverage compared with existing IR-level EDDI. The results show that FERRUM is superior over existing state-of-the-art EDDI technique in both SDC coverage and performance.

II. BACKGROUND

In this section, we first introduce the fault model and fault injection method in our study, followed by a description of working principle of EDDI. Finally, we discuss the compiler and platform we use.

A. Fault Model

We focus on single bit-flip transient hardware faults in processor computing components, including pipeline stages, arithmetic components, and load/store units, etc. We do not consider faults in the memory or caches, as we assume they have already been protected by ECC [6]. This is a common fault model used in related studies where EDDI techniques are applied [5], [8], [9], [21]. There are recent studies showing that multiple bit-flips are limited in current systems but may become a concern in the future [14], [22]. However, the majority of current work in the literature focuses on single bit-flip faults. Hence, we do so as well in this study. Exploring multiple bit-flips are our future work.

B. Choice of Fault Simulation

Fault simulation, commonly referred to as fault injection, is a procedure to replicate the fault occurrence process. It is frequently used in studies that investigate program-level fault tolerance. EDDI techniques have been also evaluated using similar methods in the past [13], [14].

Fault simulations can be done at different levels. For example, program level, architectural level, beam testing etc. Among all, beam testings are believed to be the most representative methods as they are close to how soft errors are naturally triggered [20]. However, due to the limited resources and facilities available to the public, there are only a few experts in the area who may access the technique [20], [23], [24], hence we cannot use beam testing in our work. On the other hand, micro-architectural level fault injections can be used [25], [26]. However, due to the proprietary nature of CPU designs, it remains a question whether publicly available architectural simulators can be representative to emulate the faults. Finally, program-level fault simulations can be conducted at both LLVM IR and assembly levels [18], [27]. Tools at this point are much more accessible, there are openly available fault injectors at both LLVM and assembly levels, such as LLFI [27] and PINFI [18]. There are ongoing debate on which layers are more accurate [18], [28]. To be conservative, we choose to conduct our fault injection at assembly instruction level as there is a wide acceptance and numerous precedents on injecting faults at assembly level in the closely related studies [13], [28].

C. Working Principle of EDDI

The EDDI process involves creating a duplication of an instruction and comparing the computations between the original instruction and the duplicated one. The duplication of instructions and the placement of checkers are done at compile-time, then the checker placed will be executed for checking any mismatch at runtime for error detection.

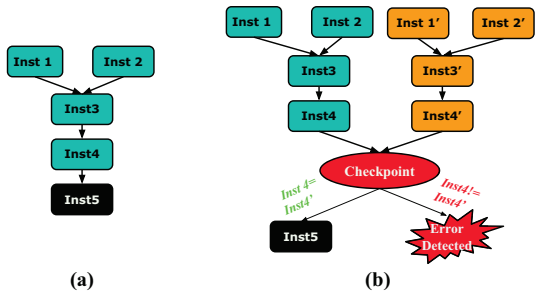

Fig. 1. High-level idea of EDDI

Figure 1 shows an example of instruction duplication technique. Figure 1-(a) represents the original program, figure 1-(b) shows the protected version after EDDI. In the example, *Inst1, Inst2, Inst3, Inst4, Inst5* are instructions in a data dependence sequence in a program, and *Inst5* denotes a synchronization point (e.g., store, control-flow branch, function call) at the end of this sequence. A data dependence sequence outlines the order in which data must be processed to ensure correct program execution. EDDI will duplicate *Inst1, Inst2, Inst3, Inst4* and insert a checkpoint before *Inst5*. If any faults happen at either original or duplicated instructions, the checker will detect the mismatch at runtime and report the error (if any).

In figure 2, we illustrate the code examples of the same program fragment at C source code and LLVM IR, showing the protection that EDDI provides. The original C code defines

a simple function *'add'* that takes two integers as arguments and returns their sum. After compiled to LLVM IR, EDDI duplicates *load* and *add* instructions, and then adds a checker after the duplication.

```
// High-level C code
int add(int a, int b) {
    return a + b;
}
```

```
define i32 @add(i32 %a, i32 %b) {
entry:
  %a.addr = alloca i32, align 4
  %b.addr = alloca i32, align 4
  store i32 %a, i32* %a.addr, align 4
  store i32 %b, i32* %b.addr, align 4
;Duplicate instruction
  %0 = load i32, i32* %a.addr, align 4
  %1 = load i32, i32* %a.addr, align 4
;Duplicate instruction
  %2 = load i32, i32* %b.addr, align 4
  %3 = load i32, i32* %b.addr, align 4
;Duplicate instruction
  %add = add nsw i32 %0, %1
  %add2 = add nsw i32 %2, %3
;Check the results
  %cmp = icmp eq i8** %add, %add2
  br i1 %cmp, label %4, label %checkBb

checkBb:
  call void @check_flag()
  br label %4

<label>:4
  ret i32 %add
}
```

Fig. 2. IR code examples of using EDDI

D. Compilation

Most existing EDDI tools available in the literature are implemented at LLVM IR level due to the availability of compiler tools in both code analysis and transformation in LLVM [8], [14], [17]. The procedure of using existing IR-level EDDI is as follows: First, the source code of the target program needs to be compiled to LLVM IR code. In this step, LLVM compiler (e.g., Clang) is used. Then an EDDI library, as a set of LLVM compiler passes, is invoked on top of the target program IR code, transforming the IR code to a protected version of IR. Finally, the protected IR code is down compiled via backend compiler to the executable.

On the other hand, it is possible to deploy EDDI at assembly level, a straightforward process is as follows: The source of target program is compiled down to assembly code, then EDDI methodology can be applied on the compiled assembly code before translating to executable. Due to the lack of openly available tools and implementations of assembly-level EDDI, most studies in the literature focus on LLVM-IR-level EDDI.

E. Platform

We target x86 ISA platform where we design and develop our proposed technique and test existing EDDI techniques. This is because x86 is one of the most popular platform [29], and thereby our choice. Other ISAs with similar features and designs may have similar designs when deploying EDDI, but we refer to the extension to other platforms as our future work.

III. FERRUM

In this section, we first describe the overall design of FERRUM, and then we dive into the technical details of the technique before we discuss the implementation.

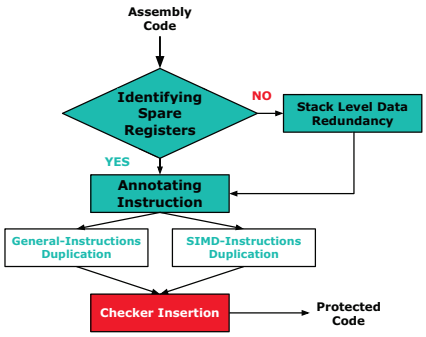

Fig. 3. FERRUM design

A. High-Level Design

We describe the overall design of our proposed technique. Recall that FERRUM aims to protect programs from hardware transient faults at assembly instruction level. The detailed steps of the technique is presented in figure 3. There are four key steps in the technique: First, FERRUM scans and examines all the registers that the target program has used, and figures out which ones are spare and available for implementing the protection. In our technique, the registers are classified into two groups. One is general-purpose registers, while the other one is SIMD registers. During the tracking, FERRUM keeps a record of the usage in both groups. At the same time, FERRUM performs instruction annotation by examining each instruction in the code, and determines if the instruction can be duplicated and checked via SIMD fashion. After FERRUM knows the spare registers and instruction information, it duplicates the instruction and places the checkers accordingly. In a nutshell, based on the availability of spare registers and the type of instructions, FERRUM duplicates instructions and places the data into available SIMD registers first before considering general-purpose registers. In the cases where there are no spare registers in either group, FERRUM leverages stack in memory at certain points of the program execution to temporarily buffer selected registers and make them available for duplication before restoring their states.

B. Components of FERRUM

1) Static Code Analysis: In this phase, FERRUM first conducts static analysis to the target code. There are two goals in this step: (1) Identifying spare registers, and (2) annotating each instruction. FERRUM does so by examining every instructions in the code, starting from the beginning.

To identify spare registers, FERRUM keeps record of the usages in both general-purpose and SIMD registers throughout the program code. After counting the usage of the registers, we check if the spare registers are enough for implementing our protection. For general-purpose registers, FERRUM requires two spare registers. This is because we need one register to protect GENERAL-INSTRUCTIONS showed in figure 4 and two registers to protect comparison instructions showed in figure 5 Whereas for SIMD registers, it demands 4 spare XMM registers for SIMD execution. These are the minimum number of spare registers for implementing FERRUM in a target program to fully leverage SIMD and register-level data redundancy. However, if the number of spare registers falls below these thresholds, FERRUM will handle the data redundancy in the protection using stack and make registers temporarily available in the protection before restoring them back to normal (more in in Section III-B4). The reason FERRUM demands 4 spare XMM registers, for example, is because we eventually need to shift 4 XMM into 2 YMM registers for comparison so on and so forth.

During instruction annotation, for each instruction, FERRUM examines whether the target instruction is suitable for using SIMD. In x86, FERRUM does so by checking if the source register is the same as the destination register. This is because if an instruction uses the same register as both source and destination, there will be no corresponding single operation for moving the data to SIMD register for execution. Consequently, a few more instructions need to be added to move the data which significantly prolongs hinders the execution time. This is a limitation of current x86 ISA support for SIMD, but future development of the ISA may address this issue. For the instructions where SIMD execution can be utilized, we name them SIMD-ENABLED-INSTRUCTIONS. For the rest, we name them GENERAL-INSTRUCTIONS. The protection strategies will be different based on the type of instructions.

2) Duplication for GENERAL-INSTRUCTIONS*:* To this end, FERRUM duplicates each instruction and places checkers for the duplication. For GENERAL-INSTRUCTIONS, FERRUM duplicates the target instruction with the same source registers and replaces the target register with one available general-purpose register, and then places a *jne* instruction for comparing the duplicated result with the original computation. A code example is shown in figure 4 for the protection of GENERAL-INSTRUCTIONS. In the example, FERRUM aims to protect *movslq* instruction. In this case, there is only one spare register is needed for duplicating and checking the results.

Figure 5 shows another example of protecting GENERAL-INSTRUCTIONS. The code example is for protecting comparison instruction. As seen, for storing the result of the *cmpl*,

```
.LBB0_3:
...
movslq  %ecx, %r10
movslq  %ecx, %rcx #original instruction
xorq    %rcx, %r10
jne exit_function
...
```

Fig. 4. Protection of GENERAL-INSTRUCTIONS

two spare registers are needed. The *cmpl* instruction changes the *rflag* register according to the result of *cmpl* instruction. However, the *rflag* register is different from other registers because its value cannot be directly and explicitly used for comparison as is done with other registers. Retrieving its value from stack requires expensive data movement, which will be inefficient. To address this issue, we propose *deferred detection* to safeguard the *rflag* register. This approach involves inserting an operation after each comparison instruction that determines the jump destination, prior to any jump instruction. This operation is designed to transfer the zero flag from the *rflag* register to a spare register. This procedure is repeated another time in order to duplicate the jump instruction and store the value into the additional spare register. It is important to note that we avoid immediate comparison of the two registers since such an operation would modify the value in the *rflag* register, voiding our effort of the protection. Instead, FERRUM identifies the basic block of code associated with the jump instruction. During the execution of the basic block, FERRUM first performs a check on the registers to detect if there is a mismatch, hence to protect comparison instruction. Note that our approach still works if there are multiple predecessors, and the trick lies in that we employ the same registers for comparison instructions. According to liveness analysis, after the check process, the register can immediately be put into new use. Our approach does not especially take indirect jump into consideration, because such cases (e.g., goto statement) are not suggested [30].

The example illustrates that after the comparison instruction, a set instruction 'sete %r11b' is appended for storing the zero flag's value. FERRUM then duplicates the instruction, storing the result of a second *cmpl* operation in '%r12b'. Subsequently, it jumps to the potential target address ('.LBB7_4' in this example) and performs an XOR check, hence protecting the *rflag* register.

3) Duplication for SIMD-ENABLED-INSTRUCTIONS*:* The main idea to protect SIMD-ENABLED-INSTRUCTIONS is that we duplicate SIMD-ENABLED-INSTRUCTIONS and shift multiple duplication and original results to SIMD registers, then compare the values at once. Since SIMD registers are larger than general purpose registers in size, a SIMD register can hold multiple results of general-purpose registers of which size are up to 64 bits. For example, a XMM register has 128 bits in width, and a YMM register holds 256 bits, whereas a general-

```
.LBB7_3:
...
cmpl  -12(%rbp), %eax
sete  %r11b #set original flag
cmpl  -12(%rbp), %eax
sete  %r12b #set duplication flag
jl  .LBB7_4
...
.LBB7_4:
xor  %r11b, %r12b #check flag value
jne exit_function
...
```

Fig. 5. Protection of comparison instruction

purpose register has only up 64 bits. Therefore, by having 4 spare XMM registers, 2 of them stores 4 computation results for original instructions and the other 2 stores 4 computation results for the duplicated instructions, we can shift them into 2 spare YMM registers and compare the two at once using SIMD execution. Note that it is also viable to leverage ZMM registers in our design, of which each has 512 bits, it depends on the underlying CPU architecture – only part of high-performance processors from Intel supports ZMM registers [31].

To implement the logic, when FERRUM imposes duplication for SIMD-ENABLED-INSTRUCTIONS, a counter is maintained in order to track how many duplications have been shifted into each SIMD register. In the case of reaching the end of a basic block, FERRUM has to check the mismatch even if the SIMD registers are not full.

```
BB1:
movq        -24(%rbp), %xmm0
movq        -24(%rbp), %rax #original Ins
movq        %rax, %xmm1
pinsrq      $1, 8(%rax), %xmm0
movq        8(%rax), %rdi   #original Ins
pinsrq      $1, %rdi, %xmm1
...
movq        -24(%rbp), %xmm2
movq        -24(%rbp), %rax #original Ins
movq        %rax, %xmm3
pinsrq      $1, 16(%rax), %xmm2
movq        16(%rax), %rdi  #original Ins
pinsrq      $1, %rdi, %xmm3
vinserti128 $1, %xmm2, %ymm0, %ymm0
vinserti128 $1, %xmm3, %ymm1, %ymm1
vpxor       %ymm1, %ymm0, %ymm0
vptest      %ymm0, %ymm0
jne exit_function
...
```

Fig. 6. FERRUM using SIMD capability

Figure 6 illustrates an example of the protection for SIMD-ENABLED-INSTRUCTIONS in *Pathfinder* benchmark. In this example, the first step involves copying the results of the target instruction and its original outcome into the lower 64 bits of the *xmm0* and *xmm1* registers, respectively. Subsequently, the result of the second instruction is shifted directly into the upper 64 bits of *xmm0* and *xmm1* registers. This same procedure is repeated for the third and fourth instructions, with their results being stored in the *xmm2* and *xmm3* registers. Once the shifts are completed, it is important to note that modifying the *xmm0* and *xmm1* registers effectively alters the lower 128 bits of the *ymm0* and *ymm1* registers. Therefore, we only need to move the contents of *xmm2* and *xmm3* into the upper 128 bits of the corresponding *ymm* registers to move all the results into one register, then place a checker to check mismatch. After this transfer is complete, a comparison operation is inserted and performed for checking mismatch at runtime using SIMD.

4) Stack-Level Data Redundancy: When encountering the situations where the number of the spare registers fall below the thresholds, FERRUM identifies the registers that are not used inside each basic block and makes them temporarily available by buffering their data onto stack, then restoring them before jumping to next basic block.

Figure 7 shows an example of this. In the example, there are no spare general-purpose registers that can be used for duplication. However, *r10* register is not used in this basic block. In this case, upon entering the basic block, FERRUM inserts a 'push %r10' instruction to push *r10* onto the stack and makes it available for using. Inside the basic block, *r10* is used for the duplication and checking. At the end of the basic block, once the checking is completed, the value of *r10* is popped from the stack through 'pop %r10' operation. This process allows for the temporary requisition of registers for instruction duplication with some extra performance overheads. We evaluate the technique in Section IV.

5) Other ISAs: As previously discussed, we implement FERRUM based on the x86 ISA due to its popularity in modern computing systems. However, it is possible to port FERRUM to other ISAs. For instance, the ARM architecture benefits significantly from the NEON SIMD instruction sets, which are optimized for efficient execution on ARM-based systems. Similarly, the AVX-512 instruction set enhances the x86 architecture by introducing ZMM registers, facilitating more efficient data backup and comparison operations. Hence, other platforms may offer additional optimization opportunities based on the characteristics of each ISA. We refer the extension to other ISAs as our future work.

IV. EVALUATION

In this section, we first introduce our experimental setup before presenting the results.

A. Experimental Setup

1) Benchmarks, Baselines, and Platform: We choose 8 benchmarks from Rodinia suite [32] in our evaluation, they are commonly used in closely related studies [14], [16], [33].

```
.LBB1_40:
push    %r10    #get temporary use
...
movslq        -68(%rbp), %r10
movslq        -68(%rbp), %rax
cmpq              %rax, %r10
jne exit_function
...
pop     %r10    #reload to previous value
```

Fig. 7. Register requisition using stack

Details of the benchmarks can be found in Table II. We first compile our benchmarks from source code to LLVM IR in order to deploy IR-based protection, which is our first baseline – the details of it will be described later. We name this baseline IR-LEVEL-EDDI.

On the other hand, we try our best to replicate a hybrid assembly-level EDDI technique based on the working principle of EDDI presented in the literature [5], [13], [34](described in Section II-C). Due to the lack of open-source code at the assembly level EDDI available in current research, we have assembled a hybrid version of EDDI through several methods. For both GENERAL-INSTRUCTIONS and SIMD-ENABLED-INSTRUCTIONS, we employ the protection method illustrated in figure 4, where each protectable instruction is immediately duplicated and checked. For comparison and branch instructions, however, we opt for delayed protection at IR level through the use of signatures [13]. The decision to implement these two instructions' protection at IR level stems from the considerable challenges associated with implementing such protections at the assembly level. Additionally, there are already existing open-source IR level protection patches designed to protect them [13]. This approach allows us to effectively implement and evaluate the EDDI system despite the challenges posed by the absence of directly comparable open-source projects. This version has no other improvement such as SIMD usage we mention above. We refer this plain assembly-level EDDI as our second baseline – HYBRID-ASSEMBLY-LEVEL-EDDI. We compile the IR to assembly code so that we can deploy HYBRID-ASSEMBLY-LEVEL-EDDI. Finally, we deploy FERRUM for the same assembly code for evaluation.

TABLE I
FERRUM AND BASELINE TECHNIQUES

	basic	**store**	**branch**	**call**	**mapping**	**comparison**
IR-LEVEL-EDDI	IR	/	/	/	/	/
HYBRID-ASSEMBLY-LEVEL-EDDI	AS_1	AS_1	IR	AS_1	AS_1	IR
FERRUM	AS_2	AS_2	AS_2	AS_2	AS_2	AS_2

Table I compares the implementation differences between FERRUM and the two baselines. IR denotes that the protection is implemented at IR level (under "basic" column), AS_1 represents the protection is implemented at assembly level without SIMD, whereas AS_2 indicates that the protection is implemented at the assembly level with SIMD utilization. Other columns indicates the assembly instruction types that whether each technique can cover or no if faults are injected to the instructions. These cases are discovered in our prior study as well [13]. We list them in the table to show the deficiency of the coverage in each technique. As seen, FERRUM can protect all types of instructions at the assembly level. Whereas HYBRID-ASSEMBLY-LEVEL-EDDI addresses the protection of branch and comparison instructions at IR level.

TABLE II
DETAILS OF BENCHMARKS

Benchmark	**Suite**	**Domain**
Backprop	Rodinia	Machine Learning
BFS	Rodinia	Graph Algorithm
Pathfinder	Rodinia	Dynamic Programming
LUD	Rodinia	Linear Algebra
Needle	Rodinia	Dynamic Programming
kNN	Rodinia	Machine Learning
kmeans	Rodinia	Data Mining
Particlefilter	Rodinia	Noise estimator

We conduct our experiment in Ubuntu 20.04 machine with an Intel Xeon processor which implements x86-64 architecture. The machine equips with 64GB RAM and GCC v5.4.0.

2) Fault Injection Methodology: In our experiment, we need to measure the SDC coverage of programs before and after each technique. We do so by conducting fault injections at assembly level as per discussed in Section II. The details of the fault injection process are described as follows: Based on our fault model, we inject single bit-flip faults to the destination register of instructions. It is a sampling process where we randomly choose a dynamically executed instruction and a random bit in the destination register of the instruction for fault injection. One fault is sampled in one program execution. In each measurement, we sample 1000 faults in each benchmark to get statistical significance. Our fault injection method is inline with other studies in the area [13], [18], [19].

3) Metrics: We choose three metrics in our evaluation in order to compare FERRUM with the two baselines:

- *SDC Coverage:* We measure SDC coverage after each protection technique is deployed. This gauges the effectiveness of a protection technique that mitigate SDCs in programs. We define SDC coverage as follows: $(SDC_{raw} - SDC_{prot})/SDC_{raw}$, where SDC_{prot} and SDC_{raw} denote program SDC probability with and without protection respectively.
- *Runtime Performance Overhead:* We measure runtime performance overhead following the deployment of each

protection technique. We measure the runtime performance overhead on average of three executions in order to minimize noises in each technique. We define runtime performance overhead as follows: $(Runtime_{prot} - Runtime_{raw})/Runtime_{raw}$, where $Runtime_{prot}$ and $Runtime_{raw}$ denote program runtime with and without protection respectively.

- *Time to execute* FERRUM*:* We measure the time to execute FERRUM. The execution of FERRUM is at compile time and is measured in wallclock time.

B. Results

To this end, we evaluate FERRUM and the baselines according to the metrics and then analyze the results.

1) SDC Coverage: Figure 10 demonstrates the SDC coverage provided by the two baselines and FERRUM. Here are our main observations from the experiments. First, FERRUM and HYBRID-ASSEMBLY-LEVEL-EDDI both provide 100% SDC coverage across all the benchmarks, whereas IR-LEVEL-EDDI cannot reach 100% SDC coverage in most of the benchmarks. In fact, among 8 benchmarks, only *kmeans* has full SDC coverage when using IR-LEVEL-EDDI protection. On average, the SDC coverage provided by IR-LEVEL-EDDI technique is 72%. In *kNN* and *Needle* benchmarks, the SDC coverages are merely 50% and 54%. This is rather ineffective protection as half of SDCs are still observed in the fault injection experiments even though the programs are protected by IR-LEVEL-EDDI.

```
1 BB3:
2 br i1 %4, label %5, label %8
```

Fig. 8. Branch Instruction at IR Level

```
1 .LBB1_4:
2 cmpl    $0, -4(%rbp)     #New FI Site
3 je      .LBB2_2
```

Fig. 9. Branch Instruction at Assembly Level

We review every scenario that leads to a loss of SDC coverage in our experiment and identify two main root-causes. Firstly, certain instructions can create potential fault injection sites when translated into assembly language, which aren't visible at IR level. This includes store, branch, and call instructions. Second, some protection that exists at IR level may become ineffective once the code is converted from IR to assembly level.

We explain this using branch instruction as an example: At LLVM IR level, the conditional branch instruction uses a true/false condition and two destinations as the perimeters. When this is turned into assembly code, the condition is usually already in the FLAGS register, so the jump instruction can just use that condition directly. But this only works if the last instruction is an *'cmp'* instruction. If not, the assembly code needs to set the EFLAG/RFLAG register with the condition before it jumps. This situation is common in protected IRs.

Figure 8 provides an example where the branch instruction is positioned at the beginning of a basic block. As a result, in figure 9, at the assembly level, a *'cmpl'* instruction is added to prepare the EFLAG/RFLAG register before it proceeds to jump to the destination address. At IR level, we do not consider branch instructions as potential sites for fault injection, but this changes at the assembly level. In assembly level fault injections, we identify faults that are introduced into the status register following the test instruction, as mentioned earlier, which can result in SDC. Similar observations on the loss of protection coverage across layers have been also reported in other recent studies [13], [19].

Note both FERRUM and HYBRID-ASSEMBLY-LEVEL-EDDI provides full coverage in all the benchmarks. That is, there is no SDC observed when injecting faults at assembly level in the programs protected by either IR-LEVEL-EDDI or FERRUM. This meets our expectation since both FERRUM and HYBRID-ASSEMBLY-LEVEL-EDDI protection provide fine-grained protection at assembly level – every possible fault injection site in assembly is duplicated and checked in FERRUM and HYBRID-ASSEMBLY-LEVEL-EDDI.

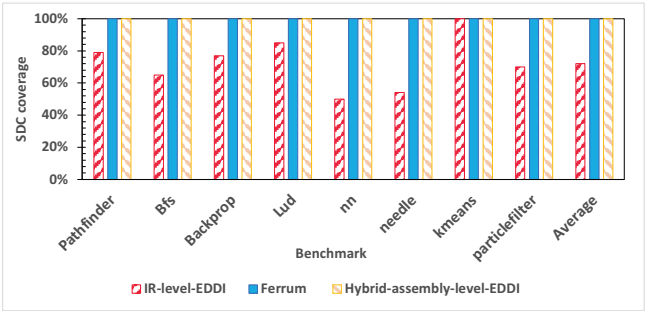

Fig. 10. SDC coverage measured with FERRUM, IR-LEVEL-EDDI and HYBRID-ASSEMBLY-LEVEL-EDDI; X-axis denotes 'benchmark', and Y-axis denotes 'SDC coverage' measured.

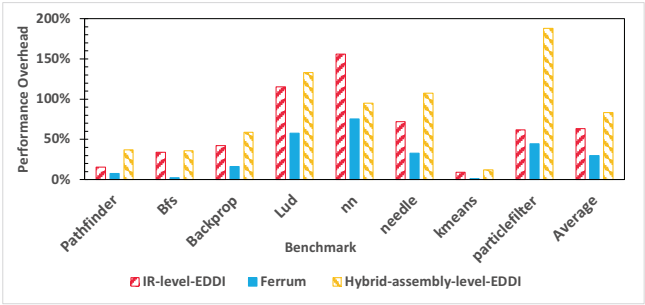

Fig. 11. Performance overhead measured with FERRUM, IR-LEVEL-EDDI and HYBRID-ASSEMBLY-LEVEL-EDDI; X-axis denotes 'benchmark', and Y-axis denotes 'runtime performance overhead' measured.

2) Runtime Performance Overhead: Figure 11 shows the runtime performance overhead incurred by deploying FERRUM, IR-LEVEL-EDDI and HYBRID-ASSEMBLY-LEVEL-EDDI respectively. As can be seen, FERRUM incurs the lowest overhead on average across all benchmarks compared with our baseline techniques. On average, the runtime performance overhead incurred by the IR-LEVEL-EDDI and HYBRID-ASSEMBLY-LEVEL-EDDI are 62.27% and 83.39%, while in FERRUM the overhead is merely 29.83%. This shows that FERRUM is significantly faster than existing IR-level EDDI and assembly-level EDDI.

In more details, we observe that HYBRID-ASSEMBLY-LEVEL-EDDI incurs higher performance overhead compared with IR-LEVEL-EDDI. This is surprising as a native implementation of the protection at assembly level is supported to be more efficient compared with an IR-level implementation. One of possible reasons why this happens could be that there are more assembly instructions generated when compiled from IR to assembly. The additional assembly instructions generated are also duplicated by HYBRID-ASSEMBLY-LEVEL-EDDI (but they do not appear at IR level protection in IR-LEVEL-EDDI), thereby incurring high overhead.

3) Execution Time: We report the time taken to execute FERRUM. Our measurement shows that FERRUM takes only 0.117 seconds on average across all the benchmarks, with a maximum of 0.196 seconds in *Particlefilter* and a minimum of 0.089 seconds in *BFS*. We find that the time taken depends on the number of static instructions in a program, as FERRUM needs to linearly scan the code and generate transformations. For example, in *Particlefilter* benchmark, the number of static instructions is 2230 while it is 406 in *BFS* benchmark.

V. RELATED WORK

EDDI has been proposed for more than two decades [35] and it became a popular technique in detecting soft errors with a low cost [5], [8], [34], especially in high-performance computing systems [36]–[42]. Reis et al. enhanced instruction duplication by integrating a software-only signature-based control-flow verification approach [34]. Lu et al. proposed SDCTune, an empirical model for predicting a program's data SDC proneness, enabling selective protection without the need for time-intensive fault injection experiments [9]. Fang et al. proposed ePVF, an enhanced Program Vulnerability Factor methodology, which provides a more discriminating metric for informing resilience techniques while reducing vulnerable bits by up to 67% and maintaining high accuracy [43]. Kalra et al. introduced ArmorAll, a lightweight and adaptive software solution for detecting soft errors in GPU, especially for accuracy-sensitive and safety-critical applications [5].

Carreira et al. introduced Xception, a sophisticated software-based fault injection and monitoring environment designed for modern and complex processors, offering a comprehensive set of fault triggers and demonstrating its potential for evaluating the dependability properties of contemporary computer systems [44]. Wei et al. assessed the accuracy of high-level software fault injection mechanisms, focusing on LLFI, an LLVM-based tool that injects faults close to the source code, and compares it to assembly-level fault injection methods to gain insights into their differences and effectiveness in quantifying application-specific resilience characteristics [18]. Qureshi et al. introduced microarchitecture-based introspection (MBI), a low-cost transient-fault detection technique that leverages otherwise wasted processing bandwidth during long-latency cache misses, making it well-suited for memory-intensive applications and resulting in only modest average IPC reductions [45]. Pham et al. introduced a comprehensive reliability modeling and prediction approach for component-based software systems that explicitly addresses factors like error propagation, fault tolerance mechanisms, and concurrent errors, offering valuable support for informed design decisions and enhanced system reliability [46].

VI. CONCLUSION

In conclusion, we propose FERRUM, an enhanced version of assembly-level EDDI with low runtime performance overhead. FERRUM is based on a hybrid version of assembly-level EDDI and upgraded by SIMD and compiler-level optimizations. Our evaluation demonstrates that FERRUM can not only achieve perfect fault coverage compared with IR-level EDDI but also has the lowest runtime performance overhead compared with both IR-level EDDI and hybrid version of assembly-level EDDI.

ACKNOWLEDGMENTS

This research was supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research (ASCR), under contract DE-SC0024559.

REFERENCES

[1] P. Shivakumar, M. Kistler, S. Keckler, D. Burger, and L. Alvisi, "Modeling the effect of technology trends on the soft error rate of combinational logic," in *Proceedings International Conference on Dependable Systems and Networks*, 2002, pp. 389–398.

[2] B. Nie, J. Xue, S. Gupta, C. Engelmann, E. Smirni, and D. Tiwari, "Characterizing temperature, power, and soft-error behaviors in data center systems: Insights, challenges, and opportunities," in *2017 IEEE 25th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2017, pp. 22–31.

[3] T. O'Gorman, "The effect of cosmic rays on the soft error rate of a dram at ground level," *IEEE Transactions on Electron Devices*, vol. 41, no. 4, pp. 553–557, 1994.

[4] D. A. G. D. Oliveira, L. L. Pilla, M. Hanzich, V. Fratin, F. Fernandes, C. Lunardi, J. M. Cela, P. O. A. Navaux, L. Carro, and P. Rech, "Radiation-induced error criticality in modern hpc parallel accelerators," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 577–588.

[5] C. Kalra, F. Previlon, N. Rubin, and D. Kaeli, "Armorall: Compiler-based resilience targeting gpu applications," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 17, no. 2, pp. 1–24, 2020.

[6] R. W. Hamming, "Error detecting and error correcting codes," *The Bell System Technical Journal*, vol. 29, no. 2, pp. 147–160, 1950.

[7] Z. Yan, H. Jiang, W. Srisa-an, S. Seth, and Y. Tan, "Leverage redundancy in hardware transactional memory to improve cache reliability," in *Proceedings of the 47th international conference on parallel processing*, 2018, pp. 1–10.

[8] I. Laguna, M. Schulz, D. F. Richards, J. Calhoun, and L. Olson, "Ipas: Intelligent protection against silent output corruption in scientific applications," in *2016 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2016, pp. 227–238.

[9] Q. Lu, K. Pattabiraman, M. S. Gupta, and J. A. Rivers, "Sdctune: A model for predicting the sdc proneness of an application for configurable protection," in *2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, 2014, pp. 1–10.

[10] M. Rebaudengo, M. Sonza Reorda, M. Torchiano, and M. Violante, "Soft-error detection through software fault-tolerance techniques," in *Proceedings 1999 IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (EFT'99)*, 1999, pp. 210–218.

[11] B. De Blaere, J. Vankeirsbilck, and J. Boydens, "Soft error detection through low-level re-execution," in *2021 5th International Conference on System Reliability and Safety (ICSRS)*, 2021, pp. 181–189.

[12] M. H. Rahman, A. Shamji, S. Guo, and G. Li, "Peppa-x: Finding program test inputs to bound silent data corruption vulnerability in hpc applications," in *SC21: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–14.

[13] Z. He, Y. Huang, H. Xu, D. Tao, and G. Li, "Demystifying and mitigating cross-layer deficiencies of soft error protection in instruction duplication," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2023, pp. 1–13.

[14] Y. Huang, S. Guo, S. Di, G. Li, and F. Cappello, "Mitigating silent data corruptions in hpc applications across multiple program inputs," in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2022, pp. 1–14.

[15] G. Li and K. Pattabiraman, "Modeling input-dependent error propagation in programs," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2018, pp. 279–290.

[16] A. R. Anwer, G. Li, K. Pattabiraman, M. Sullivan, T. Tsai, and S. K. S. Hari, "Gpu-trident: Efficient modeling of error propagation in gpu programs," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020, pp. 1–15.

[17] C. Lattner and V. Adve, "Llvm: a compilation framework for lifelong program analysis and transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, 2004, pp. 75–86.

[18] J. Wei, A. Thomas, G. Li, and K. Pattabiraman, "Quantifying the accuracy of high-level fault injection techniques for hardware faults," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014, pp. 375–382.

[19] C.-K. Chang, G. Li, and M. Erez, "Evaluating compiler ir-level selective instruction duplication with realistic hardware errors," in *2019 IEEE/ACM 9th Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS)*, 2019, pp. 41–49.

[20] F. G. Previlon, B. Egbantan, D. Tiwari, P. Rech, and D. R. Kaeli, "Combining architectural fault-injection and neutron beam testing approaches toward better understanding of gpu soft-error resilience," in *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*, 2017, pp. 898–901.

[21] G. Li, K. Pattabiraman, S. K. S. Hari, M. Sullivan, and T. Tsai, "Modeling soft-error propagation in programs," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2018, pp. 27–38.

[22] D. Tiwari, S. Gupta, J. Rogers, D. Maxwell, P. Rech, S. Vazhkudai, D. Oliveira, D. Londo, N. DeBardeleben, P. Navaux, L. Carro, and A. Bland, "Understanding gpu errors on large-scale hpc systems and the implications for system design and operation," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015, pp. 331–342.

[23] S. E. Michalak, A. J. DuBois, C. B. Storlie, H. M. Quinn, W. N. Rust, D. H. DuBois, D. G. Modl, A. Manuzzato, and S. P. Blanchard, "Neutron beam testing of high performance computing hardware," in *2011 IEEE Radiation Effects Data Workshop*, 2011, pp. 1–8.

[24] G. Bak and S. Baeg, "Failure analysis of galaxy s7 edge smartphone using neutron radiation," *IEEE Transactions on Nuclear Science*, vol. 67, no. 11, pp. 2370–2381, 2020.

[25] A. Vallero, A. Savino, G. Politano, S. Di Carlo, A. Chatzidimitriou, S. Tselonis, M. Kaliorakis, D. Gizopoulos, M. Riera, R. Canal, A. Gonzalez, M. Kooli, A. Bosio, and G. Di Natale, "Cross-layer system reliability assessment framework for hardware faults," in *2016 IEEE International Test Conference (ITC)*, 2016, pp. 1–10.

[26] J. Laurent, V. Beroulle, C. Deleuze, F. Pebay-Peyroula, and A. Papadimitriou, "Cross-layer analysis of software fault models and countermeasures against hardware fault attacks in a risc-v processor," *Microprocessors and Microsystems*, p. 102862, 2019.

[27] Q. Lu, M. Farahani, J. Wei, A. Thomas, and K. Pattabiraman, "Llfi: An intermediate code-level fault injection tool for hardware faults," in *2015 IEEE International Conference on Software Quality, Reliability and Security*, 2015, pp. 11–16.

[28] L. Palazzi, G. Li, B. Fang, and K. Pattabiraman, "A tale of two injectors: End-to-end comparison of ir-level and assembly-level fault injection," in *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, 2019, pp. 151–162.

[29] A. Heinecke, T. Auckenthaler, and C. Trinitis, "Exploiting state-of-the-art x86 architectures in scientific computing," in *2012 11th International Symposium on Parallel and Distributed Computing*, 2012, pp. 47–54.

[30] E. W. Dijkstra, "Letters to the editor: go to statement considered harmful," *Commun. ACM*, vol. 11, no. 3, p. 147–148, mar 1968. [Online]. Available: https://doi.org/10.1145/362929.362947

[31] C. S. Anderson, J. Zhang, and M. Cornea, "Enhanced vector math support on the intel®avx-512 architecture," in *2018 IEEE 25th Symposium on Computer Arithmetic (ARITH)*, 2018, pp. 120–124.

[32] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, 2009, pp. 44–54.

[33] L. Yang, B. Nie, A. Jog, and E. Smirni, "Practical resilience analysis of gpgpu applications in the presence of single- and multi-bit faults," *IEEE Transactions on Computers*, vol. 70, no. 1, pp. 30–44, 2021.

[34] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "Swift: Software implemented fault tolerance," in *International symposium on Code generation and optimization*. IEEE, 2005, pp. 243–254.

[35] N. Oh, P. Shirvani, and E. McCluskey, "Error detection by duplicated instructions in super-scalar processors," *IEEE Transactions on Reliability*, vol. 51, no. 1, pp. 63–75, 2002.

[36] X. Wei, N. Jiang, H. Yue, X. Wang, J. Zhao, G. Li, and M. Qiu, "Approxdup: Developing an approximate instruction duplication mechanism for efficient sdc detection in gpgpus," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2023.

[37] Y. Huang, Z. He, L. Li, and G. Li, "Characterizing runtime performance variation in error detection by duplicating instructions," in *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2023, pp. 730–741.

[38] B. Zhang, J. Tian, S. Di, X. Yu, Y. Feng, X. Liang, D. Tao, and F. Cappello, "Fz-gpu: A fast and high-ratio lossy compressor for scientific computing applications on gpus," in *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing*, 2023, pp. 129–142.

[39] B. Zhang, J. Tian, S. Di, X. Yu, M. Swany, D. Tao, and F. Cappello, "Gpulz: Optimizing lzss lossless compression for multi-byte data on modern gpus," in *Proceedings of the 37th International Conference on Supercomputing*, 2023, pp. 348–359.

[40] B. Zhang, B. Fang, Q. Guan, A. Li, and D. Tao, "Hq-sim: High-performance state vector simulation of quantum circuits on heterogeneous hpc systems," in *Proceedings of the 2023 International Workshop on Quantum Classical Cooperative*, 2023, pp. 1–4.

[41] S. Song and P. Jiang, "Rethinking graph data placement for graph neural network training on multiple gpus," in *Proceedings of the 36th ACM International Conference on Supercomputing*, 2022, pp. 1–10.

[42] Y. Huang, S. Guo, S. Di, G. Li, and F. Cappello, "Hardening selective protection across multiple program inputs for hpc applications," in *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2022, pp. 437–438.

[43] B. Fang, Q. Lu, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi, "epvf: An enhanced program vulnerability factor methodology for cross-layer resilience analysis," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2016, pp. 168–179.

[44] J. Carreira, H. Madeira, J. G. Silva *et al.*, "Xception: Software fault injection and monitoring in processor functional units."

[45] M. Qureshi, O. Mutlu, and Y. Patt, "Microarchitecture-based introspection: a technique for transient-fault tolerance in microprocessors," in *2005 International Conference on Dependable Systems and Networks (DSN'05)*, 2005, pp. 434–443.

[46] T.-T. Pham, X. Défago, and Q.-T. Huynh, "Reliability prediction for component-based software systems: Dealing with concurrent and propagating errors," *Science of Computer Programming*, pp. 426–457, 2015.