Characterizing Runtime Performance Variation in Error Detection by Duplicating Instructions

Yafan Huang[†][*], Zhengyang He[†][*], Lingda Li[‡], Guanpeng Li[†]
[†] University of Iowa, Iowa City, IA, USA
[‡] Brookhaven National Laboratory, Upton, NY, USA
yafan-huang@uiowa.edu, zhengyang-he@uiowa.edu, lli@bnl.gov, guanpeng-li@uiowa.edu

***Abstract*—Soft error rate has been increasing due to the shrinking size of transistors, leading to an elevated risk of catastrophic failures in modern computer systems. Error detection by duplicating instructions (EDDI) is a software-based technique to mitigate soft errors with a low runtime performance overhead and has been widely adopted in many safety- and mission-critical real-time systems such as space applications. However, these systems are commonly sensitive to runtime performance overheads the protection techniques incur. Few studies have investigated the performance of EDDI across various system designs and operational parameters, hence lacking a complete understanding in the literature. In this paper, we conduct comprehensive experiments to study the variation of EDDI runtime performance overhead and characterize the root causes. We find that there exist significant variations in performance overheads of EDDI, due to a few architectural and program-level factors. Based on the findings, we propose two practical techniques FUZZYB and CELER: FUZZYB uses an input searching technique to bound EDDI runtime performance overhead across different inputs for a given program; while CELER reduces EDDI runtime performance overheads using compiler transformation (by 25.08% reduction).**

***Keywords*—Error Resilience, Instruction Duplication, Program Analysis, Software Testing, Input Searching**

I. INTRODUCTION

Recent advances in hardware design and manufacturing technology have pushed to smaller transistor sizes, exacerbating the susceptibility of modern computer systems to soft errors [1], [2], [3]. Once a soft error occurs in a hardware component, it may corrupt the computation value, propagate in the software execution, and result in failure outcomes, such as system crashes or silent data corruptions, leading to severe social and financial catastrophes, thereby must be mitigated [4], [5], [2], [6], [7].

Error detection by duplicating instructions (EDDI) [8], a software-based technique, has been proposed to detect soft errors at software level [9], [10], [11], [6], [12]. EDDI duplicates instructions at compile time and detects mismatch at runtime if any of the two running copies is corrupted due to errors. The technique has been widely deployed in real-time systems that are safety- and mission-critical [13], [14], [15]. For example, the Stanford ARGOS project, in collaboration with Naval Research Laboratory (NRL), used EDDI in Low Earth Orbit systems for their satellites to harden the applications against strong radiation in outer space environment [14]. On the other hand, these real-time systems have multiple components working synergistically together [16]. The complex interactions, dependencies, and schedulings between components in the systems make their executions extremely sensitive to the runtime performance variations incurred by the additional protection techniques such as EDDI [17], [18]. Consequently, any timing violations in the executions due to unexpected performance variations may result in failures in the operations. For example, in the famous Therac-25 accidents [19], [20], the unscheduled subcomponents due to the timing issues result in injecting radiation doses hundreds of times higher than the normal level to the cancer patients under radiotherapy, leading to severe casualties. Unfortunately, existing studies in dependability primarily focused on the fault detection coverage perspective of EDDI [21], [22], [11], [23], [6], whereas few studies have investigated the runtime performance and its variation, leaving it an open question in the research. In our experiment, we observe that the same EDDI protection technique may incur a wide range of performance variations – from as low as 8% (e.g., FFT2 benchmark) to 203% (e.g., LBM benchmark). We find that multiple system parameters, at both software and hardware levels, play important roles and affect the performance variations of EDDI in deterministic ways. Furthermore, we observe that different program inputs also have a non-negligible impact, leading to a drastic variation of up to 68% in program executions for the same program and hardware. These findings are surprising and worrisome as they indicate that EDDI-protected systems of different designs may experience significant performance variation in the production environment where the systems run with arbitrary workloads.

To investigate and understand the runtime performance of EDDI, we conduct extensive experiments on 22 widely-used benchmarks and characterize the performance overhead in these applications. We study more than 20 possibly related factors at architectural and program levels and analyze their impacts on performance. We find that the EDDI runtime performance overhead is highly correlated to 6 dominant factors, such as the use of L1 cache, the dynamic footprints of certain instruction types, the number of operator types in duplicated instructions, etc. Based on the characterization, we propose two techniques[1] that address the vital performance issues of EDDI, FUZZYB and CELER. They can be used to assist system

[*] indicates equal contribution.

[1]Source code: https://github.com/hyfshishen/ISSRE23-FUZZYB-CELER

designers to develop safe and performal EDDI: (1) FuzzyB: A framework that efficiently bounds the runtime performance of EDDI in a given program across different program inputs. By incorporating the input searching technique, FuzzyB can locate the worst-case performance overhead in the EDDI-protected program and ensure meeting the time constraints of the system. (2) Celer: A novel compiler transformation technique that improves the EDDI runtime performance based on the insights gained in our characterization study. Celer reduces the complexity of program control-flow divergence, and significantly speeds up the program execution without affecting EDDI fault coverage, improving the time efficiency of the protection in target systems. *To the best of our knowledge, we are the first ones that comprehensively characterize the performance of EDDI, and propose practical techniques that bound and improve the performance of it.*

Our contributions are summarized as follows:

- We first measure and report the dramatic performance variation of EDDI observed from 22 benchmarks. A number of system design parameters have been studied. We quantify their relationship with the variation and identify the dominant ones.
- We propose FuzzyB to identify the upper bound of EDDI runtime performance overhead across the search space of program inputs. Our evaluation shows that FuzzyB can identify the upper bound runtime performance overhead in a timely manner, up to a speed-up of ten-fold compared with a random approach.
- Based on the understanding of the dominant design factors, we develop Celer, a control-flow optimized EDDI design, to accelerate the performance of EDDI. Compared with the original EDDI, the technique reduces the runtime performance overhead by up to 91%, with an average of 25.08% reduction.
- We conduct a case study with a real-world space application, EEKF, to further demonstrate the effectiveness of FuzzyB and Celer. The experiment shows that our techniques are highly efficient and effective in EEKF.

II. Error Detection by Instruction Duplication

EDDI is a compiler-level soft error detection technique and has been extensively studied during the past two decades [9], [11], [6], [22], [23], [10]. Since EDDI is a highly flexible and effective protection technique, it has been used to detect soft errors in mission-critical computer systems such as spacecraft [15], where even minor errors can have catastrophic consequences. The principle of EDDI can be shown in Figure 1. EDDI contains two major steps: Identifying synchronization points (❶) and EDDI code transformation (❷). Given an instruction sequence, EDDI first identifies the synchronization point *D* (❶). A synchronization point in program instruction lists denotes the end of one data dependency sequence and is usually represented as store instruction, function calls, or control-flow branches, etc. EDDI then performs code transformation to place the duplications and the related functional instructions (❷). For instance, to protect instructions *A*, *B*, and *C*, EDDI duplicates instructions by inserting *A'*, *B'*, and *C'* along with a checker before the synchronization point *D* at the compile time. If any faults occur at two independent data dependency sequences (i.e. two copies), the checker will compare the computation results between *B* and *B'* and hence detect the mismatch at runtime.

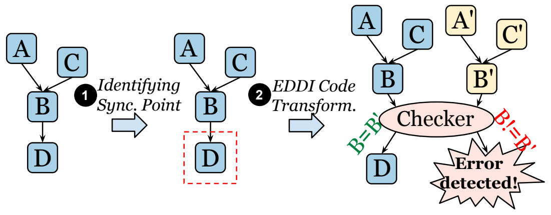


Fig. 1. EDDI principle: converting original program to program with EDDI.

Figure 2 shows an example of our EDDI implementation based on LLVM compiler infrastructure in Mantevo-HPCCG[2] application. The EDDI implementation is inline with what has been described in other recent related work [6], [23], [10], [11]. We will also use the implementation in the rest of our study. For simplicity, we refer to the terms *instruction* and *basic block* in this paper as the LLVM intermediate representation (IR) instruction and basic block, although our methodology is generic and not tied to LLVM. In Figure 2, the blue, yellow, and red part denotes the original program instructions, duplicated instructions, and functional instructions (e.g. comparison, program exit), respectively. Note that the original instructions only form one basic block without EDDI. As we can see, there are two independent data dependency sequences in the original basic block, ending up with two synchronization locations (see Lines 15 and 27). So we duplicate those two sequences separately and then insert two checkers (see Lines 8-9 and 20-21) right before the synchronizations. We implement the EDDI checker by a comparison instruction *icmp*, which compares the register values in the last instruction of two sequences. If two values are equal to each other, then no errors are detected and the program jumps to basic blocks with normal execution, such as *BB1* and *BB2*. Otherwise, the program jumps to basic blocks that report errors, such as *checkBB0* and *checkBB1*.

Similar to other state-of-the-art error resilience works [24], [22], [6], [11], [10], [4], [23], [25], our EDDI considers one single-bit-flip fault in the computation units per program execution. The reasons are two-fold: (1) Most of the soft errors that occurred on the hardware components exhibit one single-bit-flip error [7] based on an empirical study by Barcelona Supercomputing Center; (2) Soft errors in the storage components of the processors, such as memory or caches, can be effectively detected using error-correcting code (ECC) [26], [27], [28] or sampling [29] techniques. This is also one reason that EDDI technique does not duplicate synchronization instructions such as *store*. Besides, in EDDI, developers can also selectively duplicate instructions based on their reliability

[2]https://github.com/Mantevo/HPCCG

```
BB0                                  ; preds = ...
    %19 = mul nsw i32 %15, %17
    %20 = mul nsw i32 %16, %18
    %21 = load i32* %3, align 4
    %22 = load i32* %3, align 4
    %23 = mul nsw i32 %19, %21
    %24 = mul nsw i32 %20, %22
    %checker0 = icmp eq i32 %23, %24
    br i1 %checker0, <BB1>, <checkBB0>

checkBB0                             ; preds = BB0
    call void @errorDetect()

BB1                                  ; preds = BB0
    store i32 %23, i32* %local_nrow, align 4
    %26 = load i32* %local_nrow, align 4
    %27 = load i32* %local_nrow, align 4
    %28 = icmp sgt i32 %26, 0
    %29 = icmp sgt i32 %27, 0
    %checker1 = icmp eq i1 %28, %29
    br i1 %checker1, <BB2>, <checkBB1>

checkBB1                             ; preds = BB1
    call void @errorDetect()

BB2                                  ; preds = BB1
    br i1 %28, <BB3>, <BB4>
```

Fig. 2. EDDI code segment from HPCCG. The blue part denotes the original program instructions, the yellow part highlights the duplications, and the red part represents EDDI functional instructions (e.g. comparison).

target and allowable runtime performance overhead [11], [6], [10]. In this work, we focus on full duplication in EDDI to provide full fault coverage for programs subjected to our fault model.

III. Experimental Setup

In this section, we first introduce the platform and benchmarks we use in this work, then describe the methodology for how we conduct experiments to characterize and understand the EDDI runtime performance overhead variations.

A. Platform

We measure the EDDI runtime performance overhead on a Ubuntu 20.04 machine with an Intel Core i7-10700 processor (8-core/16-thread) and 64 GB RAM. The same platforms are also used in Section V and VI. The EDDI we use is implemented using the LLVM compiler (version 3.4), and follows what we describe in Section II. The EDDI method is inline with other related works about EDDI [6], [11], [10].

B. Benchmark Selection

In our experiment, we select 22 benchmarks, which are shown in Table I. The benchmarks include the ones used in several most recent error resilience studies [30], [31], [32], [11], [33]. Specifically, we include all the 3 benchmarks from [31], all 5 benchmarks from [33], 4 benchmarks from [11], and 7 benchmarks from [30]. Besides, we also include 6 commonly used HPC microkernels from NPB [34] and SPEC CPU [35] benchmark suites.

TABLE I
Details of Benchmark, where DI Count* represents dynamic instruction measured by billions.

Benchmark	**Suite**	**Domains**	**DI Count***
kNN	Rodinia	Machine Learning	19.38
Backprop	Rodinia	Machine Learning	15.06
BFS	Rodinia	Graph Algorithm	118.73
Kmeans	Rodinia	Machine Learning	31.93
Needle	Rodinia	Biology	1.59
B+tree	Rodinia	Graph Algorithm	112.91
Pathfinder	Rodinia	Dynamic Programming	170.09
LUD	Rodinia	Linear Algebra	81.93
Myocyte	Rodinia	Biology	61.32
FFT	SPLASH-2	Signal Processing	65.19
CoMD	Mantevo	Molecular Dynamics	11.75
HPCCG	Mantevo	Linear Algebra	27.30
Xsbench	CESAR	Monte Carlo Process	31.16
Blackscholes	PARSEC	Finance	6.16
BT	NPB	Linear Algebra	5.01
LU	NPB	Linear Algebra	127.68
SP	NPB	Linear Algebra	56.78
EP	NPB	Parallel Computing	21.79
FFT2	MiBench	Signal Processing	2.88
Stencil	Parboil	Stencil Operation	12.45
MCF	SPEC	Vehicle Scheduling	22.61
LBM	SPEC	Molecular Dynamics	131.67

C. Experiment Methodology

1) Terminologies: We define three terms: **(a) Runtime performance overhead (RPO):** RPO denotes the EDDI runtime performance overhead of a program, and it can be calculated by $(R_e - R_o)/R_o$, where R_e and R_o denote the execution time in an EDDI-protected program and that in the original program, respectively. The notion of RPO has been previously utilized in EDDI-related literature [36], [37]. However, we have introduced the proper noun in this context for the sake of simplification. **(b) Factor:** A parameter that has possible impacts on EDDI RPO in one program execution. **(c) Extra count per instruction (ECPI):** ECPI denotes $(P_e - P_o)/I_o$, where P_e and P_o denote the measurement for a factor, such as *L1 data cache store* execution count, in the EDDI-protected program and the original program. I_o means the number of dynamic instructions of the program execution without EDDI. We use I_o instead of P_o to normalize the increment of a factor, which reflects its impact to the overall execution more faithfully. ECPI is a concept that is widely adopted in explaining performance at the instruction level [38], [39].

2) Factor Selection: We select 22 architectural and program-level factors that are highly related and considered in conventional performance analysis to comprehend the variation of EDDI RPOs. They are listed in Table II. The selected factors can be classified into 2 categories: *instruction*, and *cache/memory*. Instruction properties represent general program characteristics, such as the number of total executed instructions and different instruction types. We also include branch misprediction information to examine whether EDDI affects the branch prediction performance. Cache/memory properties record the activities in various levels of cache/memory hierarchy. In total, we select 10 instruction and 12

TABLE II
FACTORS AND THEIR CORRECTIONS WITH EDDI RPO.

Factor	Category	Description	Measurement tool	Correlation
Dynamic-instructions	Instruction	Dynamic instruction count.	Compiler instrumentation	0.69
Standard-binary-operators	Instruction	Standard binary instruction (e.g FAdd, Add) count.	Compiler instrumentation	0.44
Floating-point-binary-operators	Instruction	Floating-point binary instruction (e.g FAdd) count.	Compiler instrumentation	0.39
Integer-binary-operators	Instruction	Integer binary instruction (e.g Add) count.	Compiler instrumentation	0.23
Logical-operators	Instruction	Logical instruction (e.g And, Xor) count.	Compiler instrumentation	-0.02
Cast-operators	Instruction	Cast instruction (e.g Trunc) count.	Compiler instrumentation	0.44
Cmp-operators	Instruction	Comparison instruction (e.g Icmp) count.	Compiler instrumentation	0.04
Basic-blocks	Instruction	Dynamic basic block count.	Compiler instrumentation	0.44
Branch	Instruction	Dynamic branch count.	Linux profiler Perf	0.13
Branch-misses	Instruction	Dynamic branch mis-prediction count.	Linux profiler Perf	-0.22
L1-dcache-loads	Cache/memory	L1 data cache load count.	Linux profiler Perf	0.63
L1-dcache-stores	Cache/memory	L1 data cache store count.	Linux profiler Perf	0.17
L1-dcache-load-misses	Cache/memory	L1 data cache load miss count.	Linux profiler Perf	-0.11
L1-icache-load-misses	Cache/memory	L1 instruction cache miss count.	Linux profiler Perf	0.42
L2-cache-instruction-hits	Cache/memory	L2 cache instruction fetch hit count.	Linux profiler Perf	0.42
L2-cache-instruction-misses	Cache/memory	L2 cache instruction fetch miss count.	Linux profiler Perf	0.29
L2-cache-data-hits	Cache/memory	L2 cache data request hit count.	Linux profiler Perf	0.22
L2-cache-data-misses	Cache/memory	L2 cache data request miss count.	Linux profiler Perf	-0.08
LLC-loads	Cache/memory	L3 cache load execution count.	Linux profiler Perf	-0.12
LLC-load-misses	Cache/memory	L3 cache load miss execution count.	Linux profiler Perf	0.04
LLC-stores	Cache/memory	L3 cache store execution count.	Linux profiler Perf	0.05
LLC-store-misses	Cache/memory	L3 cache store miss count.	Linux profiler Perf	0.01

cache/memory factors, Note that some factors are missing because of the limitation of the profiling tool used in this study, *Perf* [40]. For instance, *L1 data cache store miss* number is not available on our platform. Given that EDDI does not duplicate store instructions (Section II), the missing store miss information has a limited impact on its performance.

3) Experiment Design: To identify the key factors that affect EDDI RPOs, we design the experiments as follows.

(1) Measurement of EDDI RPOs: We measure the EDDI RPOs during the program executions. To execute the program for the measurement, we choose program inputs that produce at least 1.6 billion dynamic instructions and do not result in any reported errors. This way, the measured execution time tends to be stable with little noise on our platform. On average, the selected inputs incur 51.61 billion dynamic instructions among 22 selected benchmarks without EDDI protection and incur 86.63 billion dynamic instructions with EDDI protection. To minimize the impact of noise, the execution time we measure before and after EDDI for a program is the average execution time over three program executions. We clean up cache etc before each execution.

(2) Measurement of factor ECPI: For all the factors, we measure their ECPI on our platform. Most factors can be measured by Perf (V5.19.39), which is a profiling tool for Linux OS. For the rest factors that are related to program-level characteristics, we develop LLVM passes to measure them. Note that all the factors are measured dynamically (with the same inputs mentioned in (1)).

(3) Exploring the correlations between RPOs and factors: Finally, we explore the correlations between the EDDI RPO and each measured factor. The correlation [41] is a statistical concept that quantifies to what degree two lists of values are related to each other. A correlation value closer to 1 (or -1) denotes a stronger positive (or negative) relation and indicates a factor that may (partly) explain the implied observation.

IV. EDDI RPO CHARACTERIZATION

In this section, we characterize and understand the factors that affect the EDDI RPO. In the last column of Table II, we calculate every factor's ECPI correlation with RPOs. This section will go through each of them in more detail. We first study the EDDI RPO across different benchmarks. Among the 22 selected factors, we find 6 of their ECPI exhibit relatively stronger correlations (absolute value > 0.3) with EDDI RPOs[3]. Besides, we find the EDDI RPO in a program varies across different program inputs, and the results are also presented.

Overall, the EDDI RPO varies from 8.32% to 203.02% across the benchmarks, with an average of 80.71%, as can be seen in Figure 3. For example, benchmarks such as LBM, Backprop, and Comd reveal very high RPO in EDDI, reaching 203.02%, 145.90%, and 110.35%. On the other hand, a handful of benchmarks such as FFT2, Pathfinder, BFS have as little RPO as less than 50%.

A. Instruction Factors

The most obvious factor that causes EDDI RPOs is the increment of total dynamic instruction counts, with the highest correlation (0.69). Figure 3 illustrates measured dynamic instructions' ECPI and EDDI RPOs across the benchmarks. Among different instruction categories, we find that arithmetic instructions such as logical instructions exhibit lower correlations compared with memory accesses. This is because most of the benchmarks are memory-bound. For example,

[3]Based on a statistic study, if the correlation of two lists is greater than 0.3, they can be seen as correlated [42].

CoMD, Kmeans and HPCCG benchmarks possess more than 80% memory operators among all their dynamic instructions, whereas they are less than 10% in BFS, kNN, Pathfinder, and the computation-bounded benchmarks. Among the memory-bounded benchmarks, the CPU computing resources are often underutilized when waiting for cache/memory responses. As a result, these idle resources can be used to execute duplicated arithmetic instructions with lower and no performance penalty.

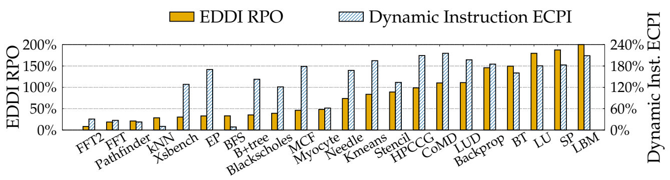


Fig. 3. EDDI RPO vs Dynamic-instructions ECPI

Figure 4 compares the computation instructions' ECPI with EDDI RPOs. Benchmarks are ordered based on their RPOs, from the lowest to highest. We observe that the ECPI of *standard-binary-operators* exhibit moderate correlations (0.44) with RPOs. We believe this has the same reason as arithmetic instructions. To further analyze binary operators, we divide them into floating-point and integer types.

Figure 5 shows the comparison of the floating-point and integer instruction ECPIs v.s. EDDI RPO. We can find that the floating-point type ECPI has a higher correlation (0.39) on the RPO compared with that of the integer type (0.23). We believe the higher correlation in floating-point instructions is because the operations of floating-point instructions take up more pipeline stages than those of integer-type ones. Since not all the original instructions are duplicated (e.g., store, jump, etc, see Section II), the duplicated floating-point instructions by EDDI contribute more towards EDDI RPO in protection.

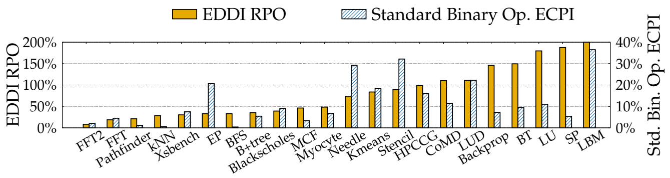


Fig. 4. EDDI RPO vs Standard-binary-operators ECPI

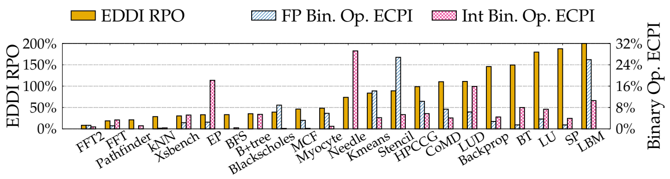


Fig. 5. EDDI RPO vs Floating-point/Integer-binary-operators ECPI

For memory operators, such as load instructions, we do not discuss them in this category since they are more related to cache/memory category which we will discuss later. In fact, duplicated memory operator counts may have strong correlations with EDDI RPOs. The reasons are two-fold: (1) Modern processors are more capable of dealing with arithmetic computations than memory accesses, due to the long latency of the latter. (2) Memory accesses adversely affect the performance of other instructions because it is difficult to schedule other instructions across memory accesses when ensuring the correctness of doing so [43]. We will analyze the detailed impact of cache/memory accesses in Section IV-B.

EDDI also introduces extra basic blocks and branch instructions due to the following reason: EDDI inserts a checker before every store instruction, and the checker introduces a branch instruction to jump to the error processing if the mismatch is detected (Section II). These newly introduced branches also break down basic blocks. As shown in Figure 6, there is a moderate correlation (0.44) between dynamic basic-block ECPI with EDDI RPOs. However, such an observation is not consistent in branch instructions. Figure 7 illustrates RPOs and incremental branch counts. This is because when the CPU encounters a branch instruction while running a program, modern CPUs will invoke branch predictors to predict the branch direction and target. As described above, the branch instruction inserted by EDDI only jumps when a soft error occurs in the program, and this situation is rare given the relatively low raw error rate. Therefore, branch predictors are very likely to make correct predictions for branches added by EDDI. This greatly reduces the impact of extra branch instructions on the EDDI RPO of the program.

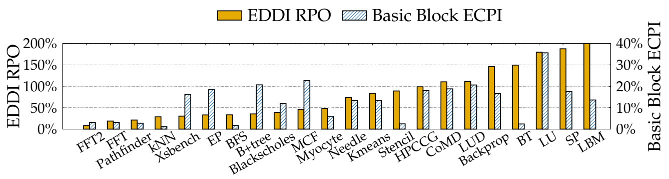


Fig. 6. EDDI RPO vs Basic-blocks ECPI

Figure 7 shows the amount of comparison instructions, branch instructions, and EDDI RPO. There is a strong correlation (0.89) between extra branch and comparison instructions since most additional comparison instructions are used to detect soft errors and therefore guide branches.

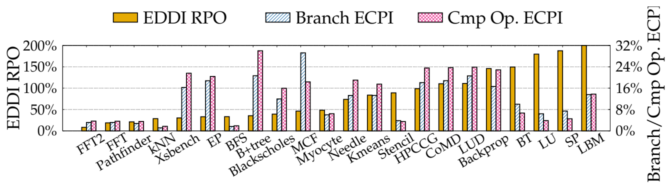


Fig. 7. EDDI RPO vs Branch/Cmp-operators ECPI

B. Cache/Memory Factors

We measure 12 cache/memory factors and explore their ECPI correlations with EDDI RPOs. These factors incorporate events that happened in all three levels of the cache hierarchy, such as *L1-dcache-stores*, *L2-cache-instruction-hits*, etc. LLC here refers to the last level cache, which is the L3 cache in our experimental platform. As shown in Figure 8, *L1-dcache-loads* ECPI has the highest correlation (0.63) with EDDI RPOs among the factors in all cache layers. And we will explain the impact of the cache from two aspects, namely the loads and misses of the cache at all levels.

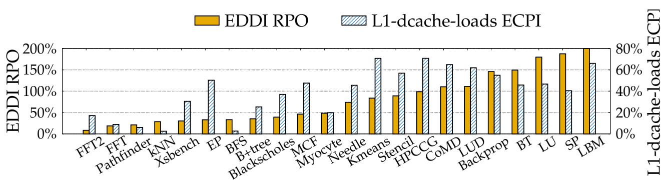


Fig. 8. EDDI RPO vs L1-dcache-loads ECPI

1) L1 Cache Loads: On one hand, the trend between EDDI RPOs and *L1-dcache-loads* ECPI across different benchmarks is pretty similar. The reason is that for a program, every time when data is stored or loaded, the L1 cache will be accessed first. Only when the data misses on L1 cache, the program will access the data to the next level of cache. Therefore, L1 caches experience the most data accesses in the cache/memory hierarchy. On the other hand, such a similar trend does not hold between EDDI RPO and *L1-dcache-stores* ECPI. Such a phenomenon is due to store instruction is not duplicated in EDDI, so the storage-related events in the cache will not contribute more in terms of RPO.

2) Instruction Fetch: We observe that instruction fetch accesses (e.g., *L1-dcache-load-misses*, and *L2-cache-instruction-misses*) have lower correlations with RPOs. Although EDDI will increase the code size and potentially cause more instruction fetch misses, modern CPUs' L1 instruction caches have sufficient capacity to accommodate the code footprint of most programs, even after applying EDDI. Therefore, EDDI does not hurt instruction fetch performance significantly.

3) L1 Data Misses and L2/LLC Accesses: We find that cache misses are not strongly correlated with EDDI RPOs. The reason is that the extra data loads/stores created by EDDI access the same addresses as the original ones. As a result, they always hit in the L1 cache and EDDI does not affect the L1 cache miss numbers. Recall that the number of L2 cache accesses is equal to the number of L1 cache misses, so it does not change with and without EDDI. Similarly, we can derive that the numbers of LLC and memory accesses do not change notably with and without EDDI. Therefore, although lower-level cache and memory accesses incur much higher latency, they have low correlations with EDDI RPOs.

C. Program Inputs

Existing works [30], [44], [32], [45], [33] have demonstrated that program inputs have significant impacts on the error propagation behaviors of a program, and hence affect SDC coverage of EDDI protections. Since error resilience and RPOs in a program are two critical parameters of EDDI, we explore the EDDI RPOs in a program across different program inputs. Of the 22 benchmarks we selected, only 16 of them provide multiple program inputs that we are able to parse – the other 6 benchmarks either lack documentation or have input formats that are not easily generated. For each benchmark used, we generate 30 random program inputs to conduct this experiment. All the generated inputs can lead to more than 1.6 billion dynamic instructions in program execution without producing any exceptions that lead to program crashes. This allows the program execution to be stable without much noise in the measurement on our platform. Since we use the same sets of benchmarks from other prior works that investigate program resiliency across multiple program inputs, we adopt the input generation method from them as well [30], [45], [46].

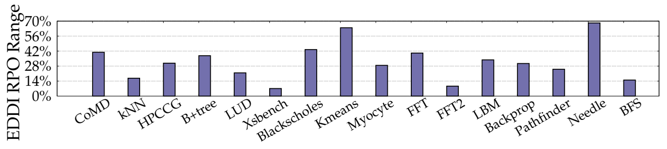


Fig. 9. EDDI RPO ranges across different program inputs

Figure 9 presents the resultant RPO ranges for each benchmark, which are calculated as the difference between the maximum RPO value and the minimum RPO value measured across the generated inputs. Here, we can observe that EDDI RPOs are input-specific, and the range is highly application-specific, varying from 7.04% in Xsbench to 68.32% in Needle. This is expected since the input impacts the execution path of a program, through different branching results and loop counts. Benchmarks with irregular computations (e.g. Kmeans) are more subject to this impact than regular ones (e.g. Xsbench).

D. Summary of EDDI RPO Characterization

In this Section, we analyze EDDI RPO variations among 10 instruction factors, 12 cache/memory factors, and program input variations. The key results are summarized as follows: **(1) 5 instruction factors have certain impacts on program EDDI RPOs**, including dynamic instruction count (0.69), standard-binary-operators (0.44), floating-point binary operators (0.39), cast operators (0.44), and dynamic basic block count (0.44). **(2) 1 cache/memory factor contributes to the variation of program EDDI RPOs**, including L1 data cache loads (0.69). **(3) Changing program inputs can lead to an obvious variation of program EDDI RPOs**, and such variation varies from 7.04% in Xsbench to 68.32% in Needle. According to the characterization results, we propose two research questions:

- **Q1**: Since EDDI RPOs are input-specific, can we bound the RPO in one program across different inputs, so that to provide an accurate EDDI performance estimation and avoid timing violation in real-time systems?
- **Q2**: Can we accelerate the EDDI RPO without losses of protection effectiveness based on the identified factors?

To answer the above two research questions, we propose two techniques FUZZYB and CELER, in order to help developers to better understand and optimize EDDI in production. FUZZYB is designed based on an input searching technique that is guided by the 6 recognized factors, to efficiently locate the highest EDDI RPO in one program among its huge input space. CELER is a faster EDDI design by control-flow optimization, which is also analyzed from the 6 dominant factors. We will explain the design details of FUZZYB and CELER in Section V and VI, respectively.

V. FUZZYB: BOUNDING EDDI RPO ACROSS PROGRAM INPUTS

As mentioned, both EDDI fault coverage and EDDI RPO are two critical parameters in the protection. Existing works have demonstrated that EDDI fault coverage varies a lot in a program across different inputs, and proposed techniques to bound it [30]. Since EDDI RPO also varies significantly across inputs (Section IV-C), one needs to bound EDDI RPO in the evaluation to understand the worst cases, take them into consideration in regard to the time and power constraints in the system design, and hence avoid any potential violations. In this use case, we propose FUZZYB that finds the program input which exposes the highest EDDI RPO in a program - we call such input *RPO-bound input* of the program.

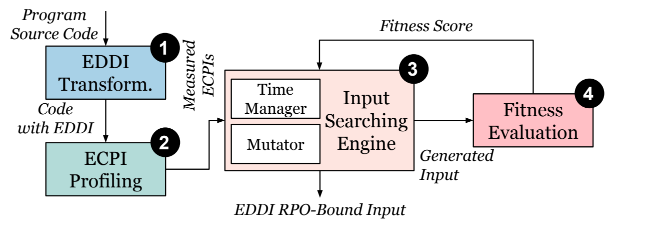


Fig. 10. Workflow of FUZZYB

A. Design Overview

Figure 10 presents the workflow of FUZZYB. FUZZYB utilizes the genetic algorithm [47] as the input searching engine and leverages the knowledge of factors identified in our analysis (Section IV) to guide the search of RPO-bound inputs for a program. FUZZYB formulates the search as an optimization problem, monitors the ECPIs of the factors, and mutates the inputs to maximize the ECPIs, in order to bound EDDI RPO. Users only need to provide the program source code, FUZZYB can generate its corresponding RPO-bound input automatically, without any interventions. FUZZYB consists of a set of LLVM passes and is driven by Python scripts. There are four major components: EDDI Transformation (❶), ECPI Profiling (❷), Input Searching Engine (❸), and Fitness Evaluation (❹).

B. Design Details

1) EDDI Transformation (❶) and ECPI Profiling (❷): Given a program source code, we perform EDDI code transformation and profile the ECPIs of the factors. We first perform static code transformation to generate the EDDI code at compile time. Then, we execute the program with EDDI and without EDDI using the measurement tool mentioned in Table II in order to profile the ECPIs.

2) Input Searching Engine (❸): After we obtain the ECPIs in the program, we feed them into the Input Searching Engine, which is an iteration-based program fuzzing engine driven by the genetic algorithm. The first iteration starts from a random input, and the input searching engine uses a mutator to slightly change one random input parameter of this input. If the selected input parameter is numerical, the input searching engines modify the value with a random number between ±10% of the current value. Otherwise, it randomly enumerates a possible value for the input parameter. The generated input will be the input of Fitness Evaluation (❹), which calculates a fitness score in turn.

3) Fitness Evaluation (❹): Fitness Evaluation is designed to calculate the generated input in the current iteration into a fitness score. We integrate knowledge obtained from Section IV as a multi-object optimization. We consider each identified high-correlation factor by weighting them with Softmax function. Specifically, given an input i, we formulate this process into Equation 1.

$$\text{Fitness}_i = \frac{1}{2}\text{RPO}_i + \frac{1}{2}\sum_{k=1}^{K}\frac{e^{\text{FC}_k}}{\sum_{j=1}^{K} e^{\text{FC}_j}}\text{Factor}_{ki} \quad (1)$$

where Fitness_i and RPO_i represent the fitness score and EDDI RPO of input i, K=6 is the number of identified high-correlation factors (absolute value > 0.3 in Table II), FC denotes the factor correlation, and Factor_{ki} represents the k-th factor of i-th input measured by dynamic profiling. Candidate inputs with the highest fitness score will survive to the next iteration, and the RPO-bound input can hence be found when the search time limit is reached. Note that there is only one program execution in each iteration, and the time overhead of the input searching engine is less than 1% compared with program execution – we measure it by the Time Manager.

C. Evaluation of FUZZYB

1) Evaluation Settings: Same as Section IV-C, we use 16 benchmarks to evaluate FUZZYB. The baseline method is the random sampling approach, which indicates executing the program with randomly sampled inputs to find the RPO-bound one – this is currently the only available approach to bound EDDI RPO [30]. We set the search time limit to 200 iterations in FUZZYB– since we find the bounded EDDI RPO will not increase for all selected benchmarks in 200 iterations.

2) Results: We evaluate FUZZYB from two perspectives: (1) the efficiency of bounding EDDI RPOs and (2) how identified factors contribute to bounding the EDDI RPOs. The first evaluates the goal of FUZZYB, whereas the other experiment demonstrates the importance of high-correlated factors in FUZZYB workflow.

Figure 11 shows the highest EDDI RPOs bounded by FUZZYB and the baseline in each benchmark. As can be seen, FUZZYB always finds the input that leads to a higher EDDI RPO given the time allowance. The only exception is the FFT benchmark, we will explain the reason in the following context. For example, in CoMD, Pathfinder, and Backprop

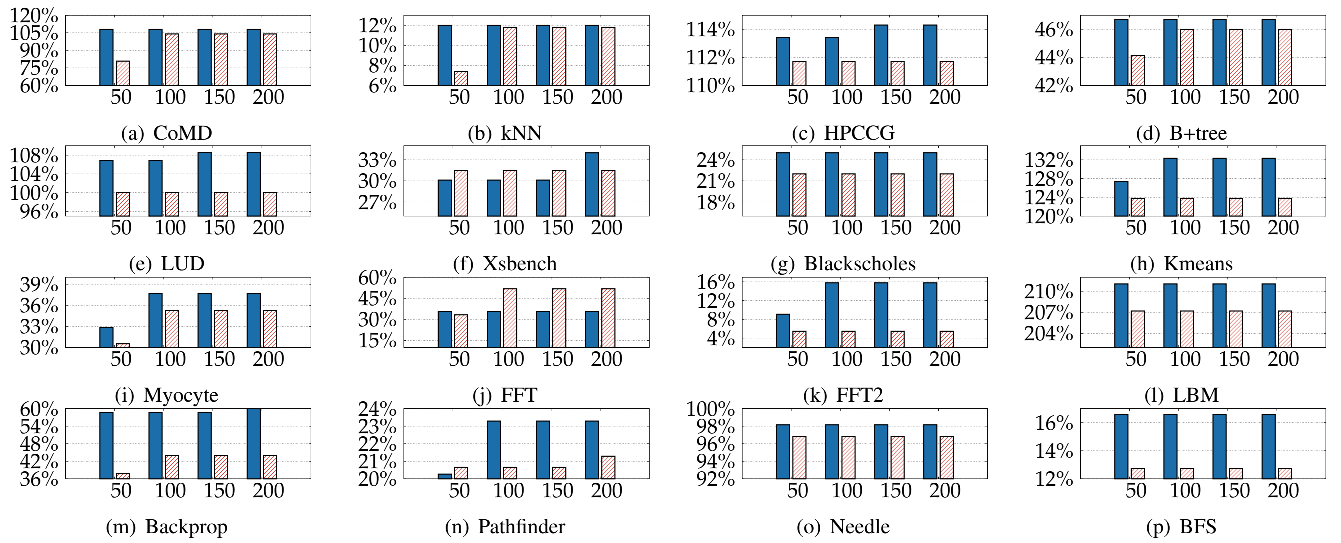


Fig. 11. The results of bounding EDDI RPOs by FUZZYB. In each subfigure, the x-axis denotes "Number of Iteration", and the y-axis denotes "EDDI RPO". The left blue bar and right red bar represent FUZZYB and Baseline.

benchmarks, FUZZYB can find the RPO-bound input with only 50, 100, and 50 iterations, whereas these numbers are 100, 200, and 100 iterations in the baseline method. In Pathfinder and Xsbench benchmarks, the baseline method can temporarily localize higher EDDI RPOs in the 50th iteration, but the bounded EDDI RPO will no longer increase in later iterations, which is far away from FUZZYB. In all, FUZZYB can efficiently find the RPO-bound input in EDDI, and it usually reaches its plateau with roughly 80 iterations on average. We also observe that the baseline method can find a higher EDDI RPO in FFT compared with FUZZYB. Due to the random nature of the baseline approach, the random sampling method may outperform FUZZYB, however, the chance is low as shown in the results.

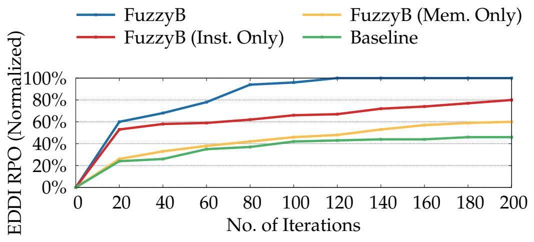


Fig. 12. The impact of identified factors in bounding EDDI RPO in FUZZYB.

Figure 12 shows our designed ablation experiments across 16 selected benchmarks in demonstrating the importance of identified factors in FUZZYB workflow. Besides FUZZYB (with all 6 factors) and baseline (with no factors and just search blindly), we add FUZZYB (Inst. Only) and FUZZYB (Mem. Only), which represent FUZZYB executed with only 5 identified instruction factors and 1 identified memory factors in the Fitness Evaluation (❹). By doing so, we can understand how those factors contribute to EDDI RPO bounding process in FUZZYB. As we can see, FUZZYB with all 6 factors can bound higher EDDI RPOs compared with the other three methods for all the iterations. FUZZYB (Inst. Only) can achieve similar EDDI RPO bounding efficiency compared with FUZZYB in the beginning 20 iterations. However, its performance then slows down considerably in later iterations due to the absence of cache/memory-level information. FUZZYB (Mem. Only) performs better in bounding EDDI RPO compared with the baseline methods, but the bounded EDDI RPOs are not as high as FUZZYB and FUZZYB (Inst. Only). Note that we also observe the EDDI RPOs bounded by FUZZYB (Inst. Only) and FUZZYB (Mem. Only) continue to increase slowly after 200 iterations – they may finally find the EDDI RPOs as high as FUZZYB with all 6 factors but obviously take much more iterations to converge. In all, upon the convergence, the bounded EDDI RPO in FUZZYB is 25.00%, 63.00%, and 117.36% higher than FUZZYB (Inst. Only), FUZZYB (Mem. Only), and baseline, respectively.

VI. CELER: ACCELERATING EDDI WITH CONTROL-FLOW OPTIMIZATION

Recall that we identified 6 factors that affect EDDI RPO variation (Section IV). Among those factors, 4 of them (including *dynamic-instructions*, *standard-binary-operators*, *floating-point-binary-operators*, and *cast-operators*) are program-level factors, which are determined during the implementation phases by developers and cannot be easily modified by users. For the cache/memory category, *L1-dcache-loads* is tied to the architecture design of the hardware, so it cannot be easily modified either. As a result, it may be possible to leverage the factor *basic-blocks* to reduce the EDDI RPO using compiler techniques, which motivates this technique.

As we mentioned in Section II, EDDI breaks original basic blocks into multiple ones by inserting the replications and functional instructions (e.g. checkers). Such operation adds additional basic blocks inevitably. Since basic block is the minimal unit to record the program controlf-low, EDDI drastically complicates the program control-flow, reducing the efficiency by shortening the window of instruction scheduling.

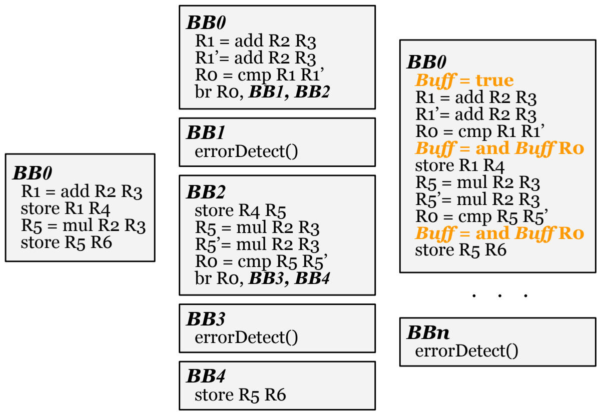


Fig. 13. Code example of original program (a), EDDI (b), and CELER (c).

In light of this, we propose a new EDDI design, called CELER, which optimizes the control-flow in EDDI. We present the detailed design and its evaluation in this section.

A. *Design of* CELER

Figure 13 illustrates how CELER works in an instruction sequence. In EDDI, the original basic block will be broken at every synchronization point (e.g., store, jump, etc, Section II). Each synchronization point also requires one extra basic block to compare the computation values between the two copies. In comparison, CELER introduces a local buffer to avoid fragmenting existing basic blocks. At the beginning of each function, CELER defines a buffer (e.g., a local variable in register) and initializes its value as true. The scope of this buffer is the entire function since all basic blocks in the same function hold a single entry and a single exit point. At each synchronization point in this function, CELER updates the buffer by a logic operator, an *and* operator, instead of breaking the current basic block into pieces. As a result, CELER only needs to check the buffer only once for a mismatch to detect errors (if any) at the end of each subroutine. By doing so CELER significantly simplifies the program control-flow, hence reducing the *basic-block* ECPI and improving the RPO.

B. *Evaluation of* CELER

We first evaluate the capability of CELER in simplifying the program control-flow. Results can be found in Figure 14. To measure the control-flow complexity, we use dynamic basic block overhead, which can be formulated as $(BB_{dup} - BB_{ori})/BB_{ori}$, where BB_{dup} and BB_{ori} represent the dynamic basic block numbers during program execution in EDDI/CELER and the original program, respectively. As we can see, CELER significantly reduces the dynamic basic block overheads on all 22 selected benchmarks. In Blackcholes, BT, and LBM, the dynamic basic block overheads in original EDDI can be as high as 629.73%, 528.84%, and 2786.50%. However, CELER optimizes those numbers into 20.11%, 2.34%, and 0.17%. In 15 of 22 benchmarks, the dynamic basic block overheads in CELER are less than 1%, indicating their program control-flow complexities are even similar to original programs. On average, the dynamic basic block overhead of CELER is 2.65%, which is only 0.83% of EDDI.

We then compare the RPOs between EDDI and CELER, and results are shown in Figure 15. We can observe that CELER significantly outperforms EDDI in most of the benchmarks. The RPO reduction can be as high as 91% in EP, whereas, in some benchmarks, such as B+tree and Pathfinder, CELER performs similarly to EDDI. This is because the instructions to be duplicated in a basic block in these benchmarks are smaller, either CELER or EDDI results in stable *Basic Block* ECPIs, hence limiting the room for improvement. Interestingly, we also observe that CELER has a negative RPO in Blackscholes - the program runs faster with CELER compared with the program execution without any protections. The dominant reason is the drastically reduced dynamic basic block overhead – only 3.19% compared with EDDI, as shown in Figure 14. Similar situations also happen in LBM, where the dynamic basic block overhead and RPO reductions are 99.99% and 44.85%. In addition, we speculate that another reason is that the original program under-unitizes hardware resources, but it gets improved with adding CELER. We also observe that CELER does not achieve reduce RPO in some benchmarks, such as LUD and Needle, even though the dynamic basic block numbers are significantly reduced. The reason can be explained below. In those benchmarks, most of the synchronization points are at the end of basic blocks, indicating that duplicating instructions in such basic blocks will not generate extra basic blocks except checker ones. In that case, CELER mainly reduces the number of checker basic blocks rather than the intervention ones. Considering other introduces computations such as buffer updates *and*, CELER compromises the execution performance compared with it in other benchmarks such as EP, Backprop, and LBM. In all, CELER is proved to be a promising solution and on average has only 62.06% RPO, which is 25.08% faster than the original EDDI.

We also evaluate the error detection efficiency of EDDI and CELER. In this case, we only focus on silent data corruption (SDC), which means the program completes its execution but produces incorrect output. SDC has been recognized as the most insidious failure type [22], [23], [6], [11]. To conduct fault injection campaigns, we randomly flip a bit in a randomly chosen instruction during the program execution and observe whether the fault injection leads to SDC or not. We then repeat the process 1,000 times in each benchmark in order to achieve statistical results. Results show that EDDI and CELER both can achieve 100% SDC coverage on 22 selected benchmarks, indicating CELER can efficiently accelerate the EDDI RPO without losing any detection effectiveness.

VII. CASE STUDY: EVALUATING FUZZYB AND CELER IN MISSION-CRITICAL SPACE APPLICATION

Beyond 22 benchmarks mentioned in Table I, we also evaluate FUZZYB and CELER with a real-world mission-critical space application. In space environments, hardware

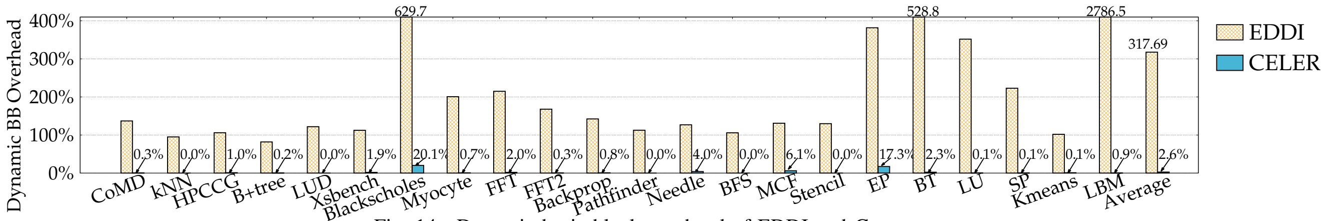


Fig. 14. Dynamic basic block overhead of EDDI and CELER.

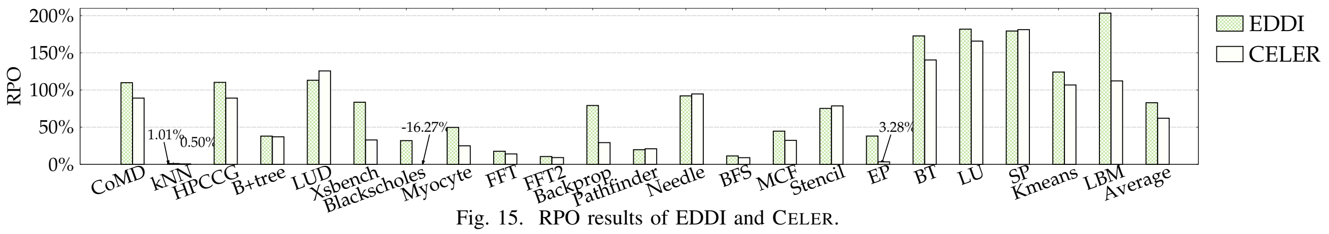
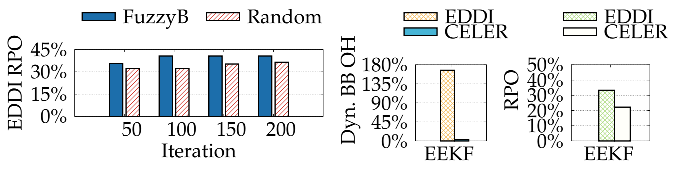


Fig. 15. RPO results of EDDI and CELER.

devices are extremely vulnerable to soft errors due to harsh cosmic radiations without atmosphere, making software-based EDDI an important protection technique [13], [14], [15]. In this case study, we evaluate our techniques with EEKF space application [48]. EEKF, designed by German Aerospace Center, is a Kalman-Filter implementation extended for embedded real-time systems. Specifically, the implementation of the core components in EEKF has been a fundamental algorithm in many spacecraft systems, such as orbit determination, altitude control etc. [49]. Thus, bounding the performance variation (by FUZZYB) and improving the EDDI performance (by CELER) are both critical and beneficial to EEKF.



(a) FuzzyB with EEKF (b) Basic Block Overhead (c) RPO

Fig. 16. Results of FUZZYB and CELER in EEKF.

Figure 16 shows the results. Similar to the evaluation section, we evaluate the effectiveness of FUZZYB in Figure 16(a) and the performance of CELER in Figure 16(b) and (c). In EEKF, FUZZYB localizes the highest RPO-bound input within 50 iterations, whereas the baseline method can reach its plateau with more than 200 iterations. Additionally, the RPO-bound input found by FUZZYB also leads to a higher EDDI RPO (40.70%) compared with the baseline method (36.50%). For EEKF with CELER, we can observe that CELER reduces the dynamic basic block overhead from 167.20% to 3.89%, hence reducing the EDDI RPO from 33.22% to 22.22%, compared with the original EDDI method. Those results are inline with what has been observed in Section V and VI, demonstrating the effectiveness of our proposed techniques in real-world space applications such as EEKF.

VIII. RELATED WORK

EDDI technique is a software-based soft error protection technique and has been proposed for more than two decades [8], [50], [13], [51]. Soon after that, it became a popular technique due to low cost and high platform portability [6], [11], [10], [12], [9], [33]. Reis et al. [9] improved EDDI by incorporating a software-only signature-based control-flow checking scheme to achieve exceptional fault coverage. Laguna et al. [11] utilized machine learning techniques to selectively duplicate the most vulnerable instructions, in order to detect soft errors at a low cost. Mahmoud et al. [12] extended EDDI to CUDA back-end compiler (ptxas) for NVIDIA GPUs. Huang et al. [32], [33] incorporated the input searching algorithm to identify incubative instructions, boosting EDDI with HPC applications across multiple program inputs. In space domains where devices are highly exposed to cosmic radiation, EDDI is also adopted due to low cost and no hardware modifications [13], [14], [15]. For example, Stanford University launched a project called CRC ARGOS project using EDDI as one of their software-implemented protection techniques [14]. Different from existing works mainly focus on fault coverage variation, our work focuses on characterizing and understanding the EDDI runtime performance overhead, and our EDDI implementation is also inline with state-of-the-art EDDI works [10], [6], [9], [11], [12].

IX. CONCLUSION

In this work, we comprehensively characterize the EDDI RPO variations, and we found that 6 hardware and software-level factors are highly correlated with EDDI RPO variations. Based on the identified factors, we propose two use cases: FUZZYB and CELER. FUZZYB can efficiently bound EDDI RPO in a program across different inputs, while CELER is a novel EDDI design to reduce EDDI RPO by compiler-level control-flow optimization. And we also implement FUZZYB and CELER in a real-world space application EEKF, the result shows that our techniques are efficient in both bounding RPO and accelerating the program's RPO.

References

[1] P. Shivakumar, M. Kistler, S. W. 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*. IEEE, 2002, pp. 389–398.

[2] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," in *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36*. IEEE, 2003, pp. 29–40.

[3] G. P. Saggese, N. J. Wang, Z. T. Kalbarczyk, S. J. Patel, and R. K. Iyer, "An experimental study of soft errors in microprocessors," *IEEE micro*, vol. 25, no. 6, pp. 30–39, 2005.

[4] G. Papadimitriou and D. Gizopoulos, "Demystifying the system vulnerability stack: Transient fault effects across the layers," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 902–915.

[5] T. Tsai, N. Theera-Ampornpunt, and S. Bagchi, "A study of soft error consequences in hard disk drives," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*. IEEE, 2012, pp. 1–8.

[6] 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.

[7] L. Bautista-Gomez, F. Zyulkyarov, O. Unsal, and S. McIntosh-Smith, "Unprotected computing: A large-scale study of dram raw error rate on a supercomputer," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2016, pp. 645–655.

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

[9] 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.

[10] 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 *Proceedings of the 2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, 2014, pp. 1–10.

[11] 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)*. IEEE, 2016, pp. 227–238.

[12] A. Mahmoud, S. K. S. Hari, M. B. Sullivan, T. Tsai, and S. W. Keckler, "Optimizing software-directed instruction replication for gpu error detection," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 842–854.

[13] P. P. Shirvani, N. Saxena, N. Oh, S. Mitra, S.-Y. Yu, W.-J. Huang, S. Fernandez-Gomez, N. A. Touba, and E. J. McCluskey, "Fault-tolerance projects at stanford crc," in *MAPLD 1999- Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference, 2 nd, Johns Hopkins Univ, APL, Laurel, MD*. Citeseer, 1999.

[14] "The argos project," in *2009 International Test Conference*, 2009, pp. 1–1.

[15] A. Spector and D. Gifford, "The space shuttle primary computer system," *Communications of the ACM*, vol. 27, no. 9, pp. 872–900, 1984.

[16] B. Zhang, Y. Huang, and G. Li, "Salus: A novel data-driven monitor that enables real-time safety in autonomous driving systems," in *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2022, pp. 85–94.

[17] A. Burns, "Scheduling hard real-time systems: a review," *Software Engineering Journal*, vol. 6, no. 3, pp. 116–128, 1991.

[18] N. Audsley, A. Burns, R. Davis, K. Tindell, and A. Wellings, *Real-time system scheduling*. Springer, 1995.

[19] N. G. Leveson and C. S. Turner, "An investigation of the therac-25 accidents," *Computer*, vol. 26, no. 7, pp. 18–41, 1993.

[20] R. Cole, "Kalman filter," https://en.wikipedia.org/wiki/Therac-25.

[21] E. Cheng, S. Mirkhani, L. G. Szafaryn, C.-Y. Cher, H. Cho, K. Skadron, M. R. Stan, K. Lilja, J. A. Abraham, P. Bose *et al.*, "Clear: C ross-l ayer e xploration for a rchitecting r esilience-combining hardware and software techniques to tolerate soft errors in processor cores," in *Proceedings of the 53rd Annual Design Automation Conference*, 2016, pp. 1–6.

[22] L. Yang, B. Nie, A. Jog, and E. Smirni, "Enabling software resilience in gpgpu applications via partial thread protection," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1248–1259.

[23] 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)*. IEEE, 2018, pp. 27–38.

[24] L. Guo, D. Li, and I. Laguna, "Paris: Predicting application resilience using machine learning," *Journal of Parallel and Distributed Computing*, vol. 152, pp. 111–124, 2021.

[25] B. Zhang, L. Yang, G. Li, and H. Xu, "Investigating the impact of high-level software design on low-level hardware fault resilience," in *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks-Supplemental Volume (DSN-S)*. IEEE, 2023, pp. 163–167.

[26] W. W. Peterson, W. Peterson, E. J. Weldon, and E. J. Weldon, *Error-correcting codes*. MIT press, 1972.

[27] M. B. Sullivan, N. Saxena, M. O'Connor, D. Lee, P. Racunas, S. Hukerikar, T. Tsai, S. K. S. Hari, and S. W. Keckler, "Characterizing and mitigating soft errors in gpu dram," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 641–653.

[28] F. Qin, S. Lu, and Y. Zhou, "Safemem: Exploiting ecc-memory for detecting memory leaks and memory corruption during production runs," in *11th International Symposium on High-Performance Computer Architecture*. IEEE, 2005, pp. 291–302.

[29] S. Silvestro, H. Liu, T. Zhang, C. Jung, D. Lee, and T. Liu, "Sampler: Pmu-based sampling to detect memory errors latent in production software," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 231–244.

[30] 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 *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–13.

[31] Z. Li, H. Menon, K. Mohror, P.-T. Bremer, Y. Livant, and V. Pascucci, "Understanding a program's resiliency through error propagation," in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2021, pp. 362–373.

[32] 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.

[33] ——, "Mitigating silent data corruptions in hpc applications across multiple program inputs," in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2022, pp. 1–14.

[34] D. Bailey, T. Harris, W. Saphir, R. Van Der Wijngaart, A. Woo, and M. Yarrow, "The nas parallel benchmarks 2.0," Technical Report NAS-95-020, NASA Ames Research Center, Tech. Rep., 1995.

[35] J. L. Henning, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[36] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. IEEE, 2006, pp. 423–432.

[37] W. Chen and E. Deelman, "Workflow overhead analysis and optimizations," in *Proceedings of the 6th workshop on Workflows in support of large-scale science*, 2011, pp. 11–20.

[38] V. Karakostas, O. S. Unsal, M. Nemirovsky, A. Cristal, and M. Swift, "Performance analysis of the memory management unit under scale-out workloads," in *2014 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2014, pp. 1–12.

[39] S. Caselli, E. Faldella, and F. Zanichelli, "Performance evaluation of processor architectures for robotics," in *Proceedings, Advanced Computer Technology, Reliable Systems and Applications*. IEEE Computer Society, 1991, pp. 667–668.

[40] Perf linux profiling tool. [Online]. Available: https://perf.wiki.kernel.org/index.php/Tutorial

[41] X.-L. Meng, R. Rosenthal, and D. B. Rubin, “Comparing correlated correlation coefficients.” *Psychological bulletin*, vol. 111, no. 1, p. 172, 1992.

[42] D. Mindrila and P. Balentyne, “Scatterplots and correlation,” *Retrieved from*, 2017.

[43] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[44] D. D. Leo, F. Ayatolahi, B. Sangchoolie, J. Karlsson, and R. Johansson, “On the impact of hardware faults–an investigation of the relationship between workload inputs and failure mode distributions,” in *International Conference on Computer Safety, Reliability, and Security*. Springer, 2012, pp. 198–209.

[45] L. Yang, “Typhoon: Enabling gpgpu application resilience estimation with different input types.”

[46] 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)*. IEEE, 2018, pp. 279–290.

[47] J. H. Holland, “Genetic algorithms,” *Scientific american*, vol. 267, no. 1, pp. 66–73, 1992.

[48] C. Schreppel, A. Pfeiffer, J. Ruggaber, and J. Brembeck, “Implementation of a c library of kalman filters for application on embedded systems,” *Computers*, vol. 11, no. 11, p. 165, 2022.

[49] R. Cole, “Kalman filter,” https://en.wikipedia.org/wiki/Kalman_filter.

[50] P. P. Shirvani, N. Oh, E. J. Mccluskey, D. Wood, M. N. Lovellette, and K. Wood, “Software-implemented hardware fault tolerance experiments: Cots in space,” in *International Conference on Dependable Systems and Networks (FTCS-30 and DCCA-8), New York (NY)*, 2000.

[51] 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 *SC23: International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2023.