

BEC: Bit-Level Static Analysis for Reliability against Soft Errors

Yousun Ko

Department of Computer Science and Engineering
Yonsei University
Seoul, Republic of Korea
yousun.ko@yonsei.ac.kr

Bernd Burgstaller

Department of Computer Science and Engineering
Yonsei University
Seoul, Republic of Korea
bburg@yonsei.ac.kr

Abstract—Soft errors are a type of transient digital signal corruption that occurs in digital hardware components such as the internal flip-flops of CPU pipelines, the register file, memory cells, and even internal communication buses. Soft errors are caused by environmental radioactivity, magnetic interference, lasers, and temperature fluctuations, either unintentionally, or as part of a deliberate attempt to compromise a system and expose confidential data.

We propose a bit-level error coalescing (BEC) static program analysis and its two use cases to understand and improve program reliability against soft errors. The BEC analysis tracks each bit corruption in the register file and classifies the effect of the corruption by its semantics at compile time. The usefulness of the proposed analysis is demonstrated in two scenarios, fault injection campaign pruning, and reliability-aware program transformation. Experimental results show that bit-level analysis pruned up to 30.04% of exhaustive fault injection campaigns (13.71% on average), without loss of accuracy. Program vulnerability was reduced by up to 13.11% (4.94% on average) through bit-level vulnerability-aware instruction scheduling. The analysis has been implemented within LLVM and evaluated on the RISC-V architecture.

To the best of our knowledge, the proposed BEC analysis is the first bit-level compiler analysis for program reliability against soft errors. The proposed method is generic and not limited to a specific computer architecture.

Index Terms—static analysis, abstract interpretation, reliability, soft errors, fault injection pruning, instruction scheduling, LLVM, RISC-V

I. INTRODUCTION

Soft errors—also known as transient hardware faults—are one of the most common threats to the reliable operation of digital devices. Soft errors temporarily alter one or more bits in hardware, thereby corrupting the execution semantics of a program. Soft errors can happen in any hardware component that is exposed to lasers, radiation [1], [2], magnetic interference [3], and temperature fluctuations [4], either accidentally or intentionally to compromise a computer system.

Soft errors can be masked without any observable effect on program execution (e.g., if after the occurrence of a soft error in a CPU register the corrupted bit is overwritten by the application logic and thus reverted to a correct state). However, soft errors that are not masked will propagate through the software stack and may lead to catastrophic system failure

by corrupting sensitive data, or altering the control flow of an application and thereby potentially granting unauthorized access to critical program paths. Mitigation of soft errors is thus an essential concern, particularly for safety-critical systems such as avionics, space, autonomous vehicles, nuclear reactor control systems, and life support devices.

Hardware-level soft error mitigation methods, such as event detection and correction (EDAC), are effective but make hardware more expensive to design, manufacture, and operate. For instance, EDAC has been reported to increase the device cost by 40% and to increase power consumption [5]. More importantly, the protection level of hardware methods is limited. For instance, EDAC can detect up to double-bit flips and correct single-bit flips, which makes software-level countermeasures indispensable [6]. A recent report cites radiati on effects as the cause for 2128 single-bit upsets in the SRAM of a satellite during a 286 day low-orbit mission [7]. The SRAM was equipped with EDAC circuitry. Fault attacks [8] are an outstanding example of soft-error exploitation, to compromise a system by injecting faults to divert the control flow of a program and thereby expose security-critical information being processed by the program.

In this work, we propose a bit-level error coalescing (BEC) analysis, to understand and improve program reliability against soft errors at bit-level. The proposed BEC analysis tracks each bit corruption in the register file and classifies the effect by its semantics at compile-time. Thus the results of the analysis can be utilized by other compiler analyses and optimizations.

BEC analyzes each bit of a register value separately and independently. This complies better with the nature of soft errors and provides more accurate analysis results compared to value-level analyses. For a soft error to propagate, the target register must contain a live value (i.e., a value that will be read in the future). But not every bit of a live value may be live. For instance, a bit-shifting operation will shift out some bits of a register while preserving other bits (albeit in different bit positions). Furthermore, BEC identifies which live bit corruptions are equivalent in their effects. Let us assume that a corrupted bit value is relocated to a new bit location via a shift operation without affecting any other operation in between. Then, the effect of a bit corruption that occurred

before the shift operation will be equivalent to the effect of a bit corruption that occurred at the new bit location after the shift operation.

We have implemented the proposed BEC analysis within LLVM 16.0.0 and demonstrated its effectiveness for eight distinctive benchmarks on the RISC-V [9] architecture. Experimental results show that the proposed bit-level analysis prunes up to 30.04% of the fault injection campaigns (13.71% on average) compared to value-level analysis, and reduces the program vulnerability by up to 13.11% through bit-level vulnerability-aware instruction scheduling (4.94% on average).

This paper makes the following contributions:

- 1) We propose BEC, the first bit-level static program analysis that exploits the semantics of operations to track and classify the effect of bit corruptions due to soft errors.
- 2) We propose an abstract bit-value analysis, which extends the scope of the analysis from sequences of instructions to global scope.
- 3) We validate the BEC analysis empirically and show its soundness.
- 4) We demonstrate the effectiveness of the BEC analysis with two use cases, fault injection campaign pruning and vulnerability-aware instruction scheduling, on eight distinctive benchmarks.
- 5) We have implemented the BEC analysis within LLVM and made it available to the public [10].

The remainder of this paper is organized as follows. We introduce the relevant background in Section II. Section III illustrates the proposed bit-level analysis with a motivating example, followed by the formal description in Section IV. Section V empirically validates the proposed method, before introducing the two use cases of the BEC analysis in Section VI-A and Section VI-B. We discuss the related work in Section VII and draw our conclusions in Section VIII.

II. BACKGROUND AND NOTATION

A program $P = \{p_0, \dots, p_{n-1}\}$ consists of $n = |P|$ program points (i.e., instructions). Each program point maintains the same set $V = \{v_0, \dots, v_{m-1}\}$ of data points (i.e., variables). The number of data points of the system that executes program P is denoted by $m = |V|$. In the case of physical hardware, m indicates the number of registers provided by the underlying hardware architecture, and m is conceptually infinite if no underlying hardware is specified (e.g., with the virtual registers of LLVM). Data points may refer to memory cells if data in memory is modeled by a compiler. Data point $v_y = [v_y^{w-1}, \dots, v_y^0]$ is of bit-width w . We use the little-endian notation for the bit representations of data points, thus, v_y^i indicates the value of the i -th least-significant bit (LSB) of data point v_y . For the sake of simplicity, we assume that all data points are of the same bit-width.

P provides the scope of the temporal structural locations of a program and V the scope of the spatial locations of the underlying hardware where soft errors may occur. We denote the Cartesian product of the two orthogonal scopes, $F = P \times$

```

1 int countYears() {
2   int res = 0;
3   for(int year=7; year>0; year--)
4     if((year%2==0) && (year%4!=0)) res++;
5   return res;
6 }

```

Fig. 1. Motivating example to count the number of years that are even but not a multiple of four, inspired by the concept of leap year.

V , as the overall *fault space*. A single *fault site* on a bit v_y^i at a program point p_x is denoted by $(p_x, v_y^i) \in F$. We define F^0 to be an empty tuple $()$.

A control-flow graph (CFG, [11]) is a directed graph $\langle P, E, e, x \rangle$, where nodes are program points $p \in P$, and edges $E \subseteq P \times P$ represent a transfer of control between program points. The unique entry and exit nodes are denoted by e and x , respectively. For program point p and CFG G , $\text{pred}(p)$ and $\text{succ}(p)$ are the sets of predecessor and successor program points. For program point p , $\text{write}(p)$, $\text{read}(p)$, and $\text{kill}(p)$ are the sets of data points written, read, and killed at p , respectively. Data points in $\text{read}(p)$ must be live before program point p , and data points in $\text{kill}(p)$ are no longer live after program point p .

We use a variation of definition-use chains [12] to connect the data points in a CFG. By $\text{def}(p, v)$ we denote the set of program points p' that define data point v and there is a CFG path from p' to p that does not re-define (kill) data point v . Similarly, by $\text{use}(p, v)$ we denote the set of program points p' that use data point v from p and there is a path from p to p' that does not re-define (kill) data point v . Note that $\text{use}(p, v)$ is sensitive to control-flow dependencies. $|\text{def}(p, v)|$ can be greater than 1, which means that data flow is not limited to static single assignment (SSA) form [13].

As the proposed BEC analysis is a bit-level analysis, we introduce a notion for the known bit values of data points at arbitrary program points. $k(p, v)$ denotes the bit values of data point v after program point p . Likewise, $k(p, v^i)$ denotes the bit value of the i -th bit from the LSB of data point v after program point p . $k(p, v^i)$ is an abstract interpretation [14] of a bit value, which is 0 or 1 if the bit is known to be zero or one in any of the temporal states of the fault site (p, v^i) , \top if it is not possible to determine the actual value of the corresponding bit at compile-time, and \perp if the value is undefined. The abstraction function is defined as $\gamma(0) = \{0\}$, $\gamma(1) = \{1\}$, $\gamma(\top) = \{0, 1\}$, $\gamma(\perp) = \emptyset$, and the concretization function is given as $\alpha(\{0\}) = 0$, $\alpha(\{1\}) = 1$, $\alpha(\{0, 1\}) = \top$, and $\alpha(\emptyset) = \perp$. The concept of $k(p, v^i)$ is comparable to *KnownBits* in LLVM and *tnum* in the Berkeley Packet Filter (BPF, [15]).

III. MOTIVATING EXAMPLE

Our motivating example counts the number of years that are even but not a multiple of four. It was inspired by the concept of leap year but simplified for the sake of exposition. For the same reason we confine the discussion of our motivating example to a 4-bit architecture. Fig. 1 depicts the source code

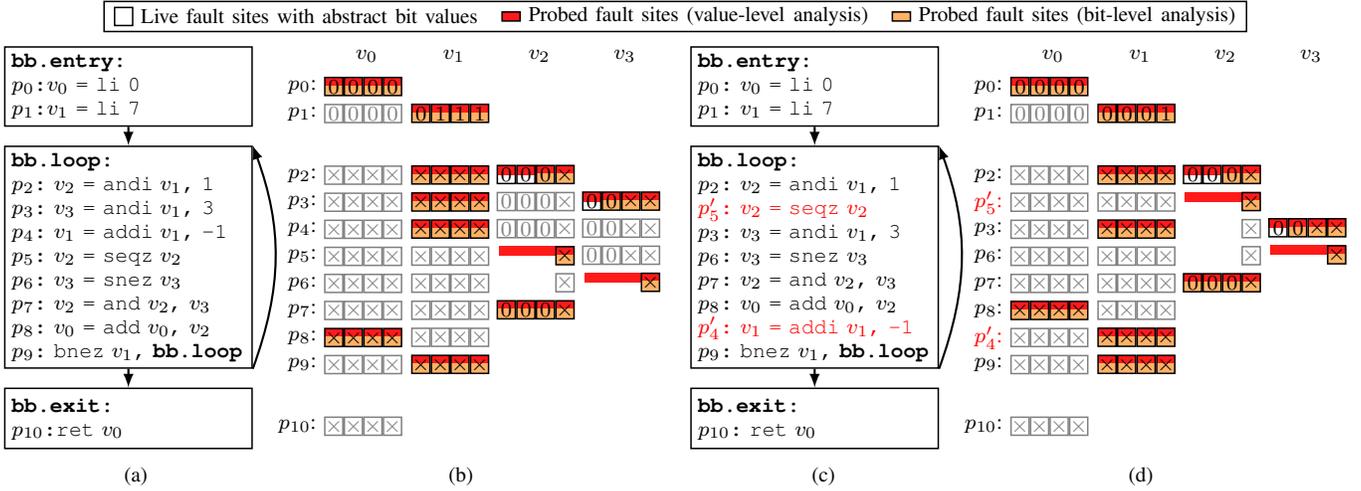


Fig. 2. (a) CFG and (b) fault sites of the motivating example from Fig. 1. With fault sites, the x-axis presents data points (variables). The y-axis refers to program points (instructions), which are labeled by their corresponding instructions from the CFG (labels p_0 – p_{10}). Live fault sites are data and program points that contain live values, depicted by white boxes with known bit values. Boxes are colored if the fault sites are identified as subjects for vulnerability tests by value-level analysis (red) or bit-level analysis (orange). The right half of the figure (c,d) depicts the motivating example after instruction rescheduling to minimize the live fault sites in bits. Note that the number of instructions to be executed and the number of fault injection runs required remain unchanged after bit-level vulnerability-aware instruction scheduling, at a reduction of the number of live fault sites by 15.4%.

of the program in C. The `for` loop (lines 3–4) iterates from 7 to 1 to accumulate in variable `res` the number of years that meet the before-mentioned condition.

Fig. 2a presents the CFG of our motivating example. The proposed analysis anticipates the bit-level semantics (i.e., side effect) of each instruction of the RISC-V target architecture [9] represented in three-address code format [11].

Fig. 2b illustrates the fault sites. Fault sites constitute the program points where the soft errors that occurred in the underlying hardware are first observed at the software level as bit flips. Soft errors can happen in any part of the hardware where machine status is stored in bits, i.e., the register file, internal flip-flops of pipelines, memory buses, memory cells etc., but soft errors must be visible to the software, for instance, as a bit corruption of the register file, to divert the behavior of the program. A bit flip of a fault site represents any soft error that happened on any hardware component and ended up corrupting a bit at the fault site. Thus the probability of observing a bit flip at a fault site due to soft errors may differ by hardware design and operational environments, but the consequence of a bit flip manifested at a particular fault site is analyzable in software. We use a single event upset model per run as it is the dominant attack model for studies from both academia and industry [16]–[18]. Bits flipped by soft errors remain on the system until overwritten.

In the presence of cycles in the CFG of a program, a single fault site can be reached multiple times during a program run, and the faults that occur across different visitations of a fault site are semantically distinguishable. For example, faults that occur at data point v_1 at program point p_2 in basic block `bb.loop` in different loop iterations are semantically different and may lead to different outcomes. For the sake of simplicity, we use four-bit unsigned values for the spatial fault

sites of our running example in Fig. 2. The LSB of a fault site is its rightmost bit.

The BEC analysis first computes the prospective bit values, $k(p_x, v_y^i)$ for all $(p_x, v_y^i) \in F$. We use \times instead of \top in the motivating examples to denote that a bit value is unknown at compile-time. Suppose we have $k(p, v) = 00\times 1$, then the value of data point v at program point p can be either 0001 or 0011. Throughout this paper, we use boxes (\square) to depict the abstract bit values of fault sites. For instance, in Fig. 2a at program point p_1 , data point v_1 is initialized to constant 7. Thus, $k(p_1, v_1) = 0111$, as shown in the second column of Fig. 2b. Inside the loop body (program points p_2 – p_9 of basic block `bb.loop`), the bits of data point v_1 are unknown because data point v_1 is an induction variable and the analysis information must hold for all possible temporal states of the loop.

Based on the analyzed bit values of fault sites, the BEC analysis identifies and classifies the effect of bit corruptions at compile-time. The static classification of bit corruptions at all possible fault sites in a program is useful for understanding the vulnerability of a program and enhancing its reliability against soft errors without additional run-time overhead. In the remainder of this section, we will outline the two use cases we have explored to demonstrate the usefulness of the proposed bit-level analysis.

A. Use Case 1: Fault Injection Pruning

A fault injection campaign is an effective technique to assess the reliability and robustness of a program or a system. It involves the systematic injection of faults into a running program to evaluate its behavior under faulty conditions. To determine the impact of a bit corruption at a given fault site, the running program is suspended at the clock cycle prior to the fault site, the bit under investigation is flipped, and the

program is resumed until the outcome becomes certain. The outcome of the program after the fault injection is compared with the golden output (i.e., the trace of events and the output obtained from a fault-free execution of a program).

The probability of a program malfunctioning in the presence of soft errors can be obtained by conducting fault injection runs on each spatial fault site of each temporal fault site—for each bit in the register file of a processor at each cycle—and counting the number of malfunctioning cases. While such an exhaustive fault injection campaign can provide valuable insights into a program’s reliability, it is an extremely resource-intensive process. The large number of faults required to obtain a statistically meaningful figure for reliability renders exhaustive fault injection infeasible for large and complex programs. A principled method for the identification and selection of a representative subset of faults to inject without sacrificing analysis precision is therefore required to accelerate fault injection campaigns.

The BEC analysis can reduce the number of fault injection sites without any loss of coverage or accuracy. In other words, the analysis results of an exhaustive fault injection campaign can be achieved by performing only a subset of the fault injection runs. As discussed in the following, the colored boxes in Fig. 2b illustrate how the proposed BEC analysis can accelerate a fault injection campaign.

Inject-on-read [18]–[20] is a method proposed to accelerate fault injection campaigns on hardware at gate-level, by injecting faults only when the fault site is read. For instance, data point v_0 in Fig. 2 is a return value that accumulates the number of the years of interest, thus, it is live throughout the lifespan of the function. But fault injection runs on data point v_0 are required only at program point p_8 per loop iteration, and the results of the fault injection runs at program point p_8 are identical to the fault injection runs performed at any program point after the previous program point that accessed the variable, which is program point p_8 of the previous loop iteration. The inject-on-read method is efficient, yet its analysis is performed on *values*. In Fig. 2b we marked the fault sites in red for which fault injection is required by the inject-on-read method.

The proposed bit-level analysis can further identify which bits *within* a value require a fault injection run. For example, data point v_2 keeps the result of instruction p_2 and p_5 in Fig. 2b, which encodes the condition `year%2 == 0` in line 4 of Fig. 1. After the execution of instruction p_2 , which masks out all but the LSB of data point v_2 , it holds that $k(p_2, v_2) = 000\times$. Knowing that instruction `seqz` at program point p_5 tests v_2 for zero, we note that any corruption that flips any one of the bits v_2^1, v_2^2 , or v_2^3 from 0 to 1 will result in the same negative test result. Consequently, only one fault injection is required among the bits v_2^1, v_2^2 , and v_2^3 at program point p_2 . As indicated by the orange color for data point v_2 at program point p_2 in Fig. 2b, our bit-level analysis injects only bit v_2^1 , whereas the prior approach injects all three bits. Because program point p_2 is part of the loop, the savings achieved by our analysis pertain to each temporal fault site

(i.e., one per loop iteration) of program point p_2 . Unlike our running example, the registers of contemporary processors are 32- or even 64-bit wide. Capitalizing on those two facts, our bit-level analysis yields noticeable gains (as discussed in Section VI-A).

Not only can the proposed bit-level analysis classify which fault sites share an identical response to soft errors, it can further analyze which fault sites are dead, meaning that bit corruption on such fault sites are known to be ineffective. Fault sites (p_5, v_2^1) , (p_5, v_2^2) , and (p_5, v_2^3) are dead in Fig. 2b because a corruption of any of those bits will be masked by the `and` operation at program point p_7 .

The fault sites required to be probed by fault injection runs by the proposed BEC analysis are marked by orange boxes in Fig. 2b, in contrast to the red boxes which are fault sites required to be probed by value-level analysis. With the motivating example in Fig. 2 in conjunction with the given loop bounds, the number of fault injection runs required by value-level analysis is 288[†], whereas the number of fault injection runs required by the proposed bit-level analysis is 225[‡]. Thus, the number of fault injection runs saved is $1 - \frac{225}{288} = 21.8\%$.

B. Use Case 2: Bit-level Vulnerability-aware Instruction Scheduling

The size of the spatial and temporal fault surface for soft errors is one of the main metrics to determine a program’s vulnerability. If Program A completes a task in the same number of cycles as Program B, but requires more hardware resources, then Program A is more vulnerable to soft errors because of its larger spatial fault surface than Program B. Similarly, if Program A takes longer than Program B to complete the same task with the same amount of resources, then program A is more susceptible to soft errors regarding its temporal fault surface. Thus, it is important to minimize the number of instructions and hardware resources required to complete a task to improve the overall reliability of a program.

The fault surface of a program can be determined by the number of live fault sites in data for every program point executed by a program run. In Fig. 2b, the number of live fault sites for a program run is 681^{††}. As in the case of data point v_2 at program points p_5 and p_6 , and data point v_3 at program point p_6 in Fig. 2b, dead fault sites identified by the proposed bit-level analysis can be exploited to reduce the overall vulnerability of the program by rescheduling of instructions.

In the example, data points v_0 and v_1 are live at almost every program point. As a result, rescheduling instructions would not decrease the number of fault sites for v_0 or v_1 on any program point. However, data points v_2 and v_3 are temporary within each iteration and only carry one fault site per program point after program points p_5 and p_6 in Fig. 2a have been executed. Thus, there is room for reducing fault

[†] $4 + 4 + 7 \times (4 + 4 \times 4 + 3 \times 4 + 2 \times 4) = 288$

[‡] $4 + 4 + 7 \times (4 + 4 \times 4 + 2 + 1 + 4 + 3 + 1) = 225$

^{††} $3 \times 4 + 7 \times (8 \times 4 + 8 \times 4 + 4 \times 4 + 2 \times 1 + 3 \times 4 + 1) + 4 = 681$

sites by instruction scheduling on these two variables. Fig. 2c shows the modified sequence of instructions and Fig. 2d shows its fault sites of the same motivating example from Fig. 1 after instruction rescheduling to minimize the live fault sites.

The number of fault sites per program point is reduced from four to one after program point p'_5 for data point v_2 and program point p_6 for data point v_3 in Fig. 2c. Thus, these instructions are executed as early as possible in Fig. 2c. Temporary variables are retired as early as possible for the same purpose. With the new sequence of instructions, the total number of fault sites of the program is reduced by 15.4% ($= 1 - \frac{576}{681}$). Note that the number of instructions to be executed and the number of fault injection runs required remain unchanged after instruction rescheduling.

IV. BIT-LEVEL ANALYSIS FOR RELIABILITY

To classify the effect of soft errors at fault sites, we introduce the notion of *fault index*, denoted by $s((p_x, v_y^i))$. A fault index labels the effect induced by a soft error at fault site $(p_x, v_y^i) \in F$. We use s_0 to denote the intact execution of the program, and $s_0 = s(F^0) = s(())$. The set of fault indices S is defined as $S = \{s((p_x, v_y^i)) \mid (p_x, v_y^i) \in F\} \cup \{s_0\}$. For a given program, if bit flips at two distinct fault sites (p, u^i) and (q, v^j) have the same effect on program execution, we consider these fault sites as *equivalent*. We employ equivalence relations [21] to represent the equivalence of fault indices based on the equivalence of their associated fault sites in terms of the underlying program semantics.

An equivalence relation $R = S/\sim_R$ constitutes the set of all equivalence classes over S induced by a binary relation \sim_R . We write $s_x \sim_R s_y$ for $s_x, s_y \in S$, denoting that the fault indices s_x and s_y are in the same equivalence class under R . Hence, the effects of faults at their associated fault sites are equivalent according to the above definition.

The equivalence class of s_x under \sim_R is defined as $[s_x]_R = \{s_y \in S \mid s_x \sim_R s_y\}$. For instance, if $[s_x]_R = \{s_x, s_y\}$, it holds that $[s_x]_R = [s_y]_R$. We define the merge of two equivalence classes as $R[[s_x]_R \cup [s_y]_R] = ((R \setminus [s_x]_R) \setminus [s_y]_R) \cup \{[s_x]_R \cup [s_y]_R\}$ for any $s_x, s_y \in S$. By abuse of notation, we merge arbitrary equivalence classes as $R[X] = R[\bigcup_{x \in X} [x]_R]$ for any $X \subset S$.

The proposed BEC analysis computes a safe approximation of the equivalence relation S/\sim_R over the fault indices S of a program. If the analysis determines that $[s((p, v^i))]_R = [s((q, w^j))]_R$, then the effects of soft errors at fault sites (p, v^i) and (q, w^j) are known to be equivalent. Conversely, if the analysis fails to establish the equivalence of two fault sites wrt. soft errors, they may still be equivalent but cannot be shown so within the analysis approximations (this is further discussed in Section V). BEC employs the abstract bit values of a fault site and is thus conducted in two steps: (1) a global abstract bit-value analysis (Section IV-A), which is a forward data-flow analysis at the granularity of the individual bits of data points, and (2) an analysis to coalesce fault indices (Section IV-B), which is a backward data-flow analysis

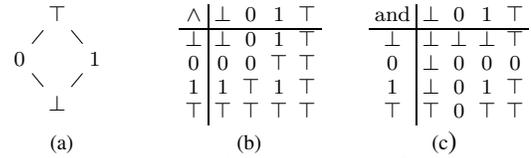


Fig. 3. Bit-level analysis: (a) lattice representation of bit values, (b) meet operator, and (c) bit-wise and operator.

Algorithm 1: Bit-Value Analysis

Input: program point $p \in P$ and data point $v \in V$
Output: $k(p, v)$, the abstract bit-value of v after the execution of p

```

1 forall  $u \in \text{read}(p)$  do
2   forall  $o \in \text{def}(p, u)$  do
3     for  $i \leftarrow \{0, \dots, w-1\}$  do
4        $k(p, u^i) \leftarrow k(p, u^i) \wedge k(o, v^i)$ 
5 forall  $v \in \text{write}(p)$  do
6   for  $i \leftarrow \{0, \dots, w-1\}$  do
7      $k(p, v^i) \leftarrow \text{op}_p(\{k(p, u) \mid u \in \text{read}(p)\})$ 

```

that identifies and classifies the corruption of the program semantics due to soft errors based on bit values.

A. Global Abstract Bit Value Analysis

To analyze the effects of soft errors at the granularity of individual bits, we first identify the bit values of all data points i.e., $k(p_x, v_y^i), \forall (p_x, v_y^i) \in F$, by performing a forward data-flow analysis across the entire program. Our analysis is inspired by a value-level constant propagation algorithm by Wegman and Zadek (SC, [22]), which we extend to bit-level analysis. We deliberately locate our analysis at a late stage within the pass sequence of the LLVM backend to benefit from target-specific strength reduction optimizations that lower arithmetic operations to bit-level operations and thereby increase the opportunity for the application of our analysis. SSA form is already deconstructed at this stage, and our analysis computes $\text{use}(p_x, v_y^i), \forall (p_x, v_y^i) \in F$ as defined in Section II. LLVM provides a bit-level analysis named *KnownBits* for straight-line sequences of code. We extend this analysis to support inter-basic-block (global) data flow.

Fig. 3a depicts the lattice representation of bit values employed with the analysis: \perp (undefined) is used for a bit at a particular program point that has not seen an assignment yet; 0 (or 1) is used for a bit of a data point where it is certain that the bit-value will be zero (or one) on all paths to the respective program point that have been considered so far, and \top (unknown and overdefined) denotes a bit value that cannot be determined at compile-time. For instance, a bit is unknown and overdefined if the bit value is zero on some paths (or loop iterations) and one on others. For each bit, the computed information can only raise in the lattice.

Algorithm 1 describes the analysis for a given program point p and data point v . Because of join points in the un-

Algorithm 2: Fault Index Coalescing Analysis

Input: a program $P = \{p_0, \dots, p_{n-1}\}$, a set of data points $V = \{v_0, \dots, v_{m-1}\}$, and bit-width w

Output: A set of equivalence classes $R = S/\sim_R$

```
/* Initialization */
1  $R \leftarrow \{\{s_0\}\}$ 
2 forall  $(p, v^i) \in F$  do
3   if  $v \in \text{write}(p) \vee v \in \text{read}(p)$  then
4     if  $v \in \text{kill}(p)$  then
5        $[s_0]_R \leftarrow [s_0]_R \cup \{s((p, v^i))\}$ 
6     else
7        $R \leftarrow R \cup \{s((p, v^i))\}$ 
/* Iterative coalescing */
8 while  $R$  is updated do
/* Intra-instruction coalescing */
9   forall  $p \in P$  do
10     $R'_p \leftarrow \text{intra\_instr\_coalescing}(p, V, w, R)$ 
/* Inter-instruction coalescing */
11   forall  $(p, v^i) \in F$  do
12     $R \leftarrow R \cup \bigcap_{q \in \text{use}(p, v)} [s((q, v^i))]_{R'_q}$ 
```

derlying CFG, multiple definitions may reach p for any of the data points it reads. The purpose of the loop starting at line 2 is to merge all definitions that reach data point $u \in \text{read}(p)$. The innermost loop iterates over all the bits of data point u . It uses the meet operator (\wedge) from Fig. 3b. For instance, the meet of definitions zero and one sets the corresponding bit to overdefined, i.e., $\wedge(0, 1) = \top$.

In case program point p computes values (e.g., the bitwise and of two values), lines 6–7 in Algorithm 1 iterate over all bits in the corresponding data point v to perform the computation (op_p in line 7) of the program point in the abstract domain of the lattice. As an example, Fig. 3c provides the definition of the bit-wise and operation. The definition of an operation ensures that the abstract value of a bit can only move upward in the lattice. We note that the analysis permits any number of values computed at a program point (line 5 in Algorithm 1), which is a generalization of the three-address code used in the motivating example in Section III.

B. Bit-level Fault Index Coalescing Analysis

The fault index coalescing analysis is a backward data-flow analysis that (1) identifies which fault sites mask soft errors and (2) classifies which fault sites result in equivalent program semantics when corrupted by a bit flip.

Algorithm 2 describes the fault index coalescing algorithm, which assigns fault indices $s \in S$ the same equivalence class if the effects of faults occurring at those indices are identical (equivalent) according to the semantics of the underlying program. The algorithm returns an equivalence relation $R = S/\sim_R$.

The loop from line 2 in Algorithm 2 initializes equivalence relation R for each data point v that is accessed (i.e., read or written) at program point p . If the accessed data point v is not live after program point p , any faults occurring at v after p will be overwritten and masked. Soft errors at masked fault sites do not alter the program semantics. Thus masked fault sites are assigned to the equivalence class s_0 (line 5 in Algorithm 2). For fault site $(p, v^i) \in F$, if data point v is accessed at and live after program point p , a new equivalence class is assigned to the fault site (line 7 in Algorithm 2). Note that data points that are not accessed at a program point may be live and susceptible to soft errors, but the equivalence relation considers those only that are accessed at the program point. This is because the semantics of a program can only be changed by reading a corrupted value regardless of when the corruption happened. Thus we postulate that the effect of any faults that occurred at a data point are the same until the program reaches the program point that reads the data point.

After initialization, equivalence relation R is a set of singletons of all fault sites of F . It is interpreted as no live fault sites are identified to be equivalent nor masked. This is sound but not necessarily precise. The analysis results are subsequently refined by coalescing equivalence classes within instructions (intra-instruction coalescing, line 10 in Algorithm 2) and across instructions (inter-instruction coalescing, line 12 in Algorithm 2), in an iterative manner.

Intra-instruction coalescing is depicted in Algorithm 3 for a selection of RISC-V instructions [9], but the proposed method is general and applicable to any other instruction set architecture (ISA). During intra-instruction coalescing, equivalence classes are merged based on the semantics of the operation of a program point, applied to the abstract bit values of the accessed data points. For instance, for program point p with bitwise operation $z = \text{and } x, y$ and a bit of operand x known to be zero, i.e., $k(p, x^i) = 0$ (line 22 in Algorithm 3), a soft error carried to fault site (p, y^i) is masked as a result of the `and` operation. Thus, the equivalence class $[s((p, y^i))]_R$ is merged with $[s_0]_R$ (line 23 in Algorithm 3). Conversely, if bit x^i of operand x is known to be one (line 24 in Algorithm 3), a fault on the corresponding bit y^i will be propagated to the result bit z^i . Thus, these two equivalence classes are merged (line 25 in Algorithm 3). Lines 18–21 in Algorithm 3 follow by commutativity of the `and` operation.

A few instructions in our analysis are oblivious to the abstract bit values of their operands. Instructions `mv` and `xor` belong to this category, and coalescing is conducted unconditionally. With `xor` (line 5 in Algorithm 3), any soft errors at (p, x^i) or (p, y^i) are propagated to (p, z^i) after the operation. Thus, the equivalence classes $[s((p, x^i))]_R$ and $[s((p, y^i))]_R$ are merged with $[s((p, z^i))]_R$ (lines 6 and 7 in Algorithm 3).

For brevity, several utility functions have been introduced in Algorithm 3. Function $\text{min}(p, v)$ returns the minimum possible value of data point v at program point p considering $k(p, v)$. Function $\text{eval}(p, v^i)$ (partially) evaluates the instruction at program point p based on the abstract bit values of the operands,

Algorithm 3: Intra-instruction Coalescing

Input: a program point $p \in P$, a set of data points $V = \{v_0, \dots, v_{m-1}\}$, bit-width w , and a set of equivalence classes $R = S/\sim_R$

Output: A set of equivalence classes $R' = S/\sim_{R'}$

```
1  $R' \leftarrow \emptyset, I \leftarrow \{0, \dots, w-1\}$ 
2 forall  $i \in I$  do
3   if  $p : z = \text{mv } x$  and  $x, z \in V$  then
4      $R' \leftarrow R[[s((p, x^i))]_R \cup [s((p, z^i))]_R]$ 
5   else if  $p : z = \text{xor } x, y$  and  $x, y, z \in V$  then
6      $R' \leftarrow R[[s((p, x^i))]_R \cup [s((p, z^i))]_R]$ 
7      $R' \leftarrow R[[s((p, y^i))]_R \cup [s((p, z^i))]_R]$ 
8   else if  $p : z = \text{or } x, y$  and  $x, y, z \in V$  then
9     if  $k(p, y^i) = 0$  then
10        $R' \leftarrow R[[s((p, x^i))]_R \cup [s((p, z^i))]_R]$ 
11     else if  $k(p, y^i) = 1$  then
12        $R' \leftarrow R[[s((p, x^i))]_R \cup [s_0]_R]$ 
13     else if  $k(p, x^i) = 0$  then
14        $R' \leftarrow R[[s((p, y^i))]_R \cup [s((p, z^i))]_R]$ 
15     else if  $k(p, x^i) = 1$  then
16        $R' \leftarrow R[[s((p, y^i))]_R \cup [s_0]_R]$ 
17   else if  $p : z = \text{and } x, y$  and  $x, y, z \in V$  then
18     if  $k(p, y^i) = 0$  then
19        $R' \leftarrow R[[s((p, x^i))]_R \cup [s_0]_R]$ 
20     else if  $k(p, y^i) = 1$  then
21        $R' \leftarrow R[[s((p, x^i))]_R \cup [s((p, z^i))]_R]$ 
22     else if  $k(p, x^i) = 0$  then
23        $R' \leftarrow R[[s((p, y^i))]_R \cup [s_0]_R]$ 
24     else if  $k(p, x^i) = 1$  then
25        $R' \leftarrow R[[s((p, y^i))]_R \cup [s((p, z^i))]_R]$ 
26   else if  $p : z = \text{shr } x, y$  and  $x, y, z \in V$  then
27     if  $i - \min(p, y) < 0$  then
28        $R' \leftarrow R[[s((p, x^i))]_R \cup [s_0]_R]$ 
29     else if  $y$  is constant and  $i - y \geq 0$  then
30        $R' \leftarrow R[[s((p, x^i))]_R \cup [s((p, z^{i-y}))]_R]$ 
31   else if  $p : z = \text{shl } x, y$  and  $x, y, z \in V$  then
32     if  $i + \min(p, y) \geq w$  then
33        $R' \leftarrow R[[s((p, x^i))]_R \cup [s_0]_R]$ 
34     else if  $y$  is constant and  $i + y < w$  then
35        $R' \leftarrow R[[s((p, x^i))]_R \cup [s((p, z^{i+y}))]_R]$ 
36 if  $p : [\text{slt}|\text{beq}|\text{bne}|\text{bge}|\text{blt}] x, y$  and  $x, y \in V$  then
37   forall  $i \in I, j \in \{0, \dots, i-1\}, v \in \{x, y\}$  do
38     if  $\text{eval}(p, v^i) = \text{eval}(p, v^j)$  then
39        $R' \leftarrow R[[s((p, v^i))]_R \cup [s((p, v^j))]_R]$ 
```

assuming that a soft error occurred at fault site (p, v^i) . If p is a branch instruction, the result of the evaluation is the target branch taken by the instruction.

The temporary equivalence relation R' is introduced to defer the merge of the equivalence classes until all fault sites have been visited during inter-instruction coalescing. Equivalence class $[s((p, v^i))]_R$ is merged and updated only if $[s((p, v^i))]_{R'_q}$ for all $q \in \text{use}(p, v^i)$ agree (line 12 in Algorithm 2) for a given $(p, v^i) \in F$.

The iterative fault index coalescing analysis terminates when no further update of the equivalence classes in R occurs (line 8 in Algorithm 2), thereby reaching a fixed point [11]. By Knaster-Tarski's fixed point theorem [23], any monotone function on a complete lattice admits a least fixed point. The collection of all equivalence relations on a set forms a complete lattice [24], and the fault index coalescing analysis is monotonic: it is performed backward along the dependency edges, from $\text{write}(p)$ to $\text{read}(p)$ and from $\text{use}(p, v)$ to $\text{def}(q, v)$. Thus, the iterative fault index coalescing algorithm is guaranteed to terminate.

C. Coalescing Example

Fig. 4a presents the initial fault indices of our coalescing example in red solid boxes. Fault indices are represented by integer identifiers such as $s((p_0, a^0)) = 1$ and $s((p_5, v_8^3)) = 24$. Fault index identifier 0 is reserved for the intact semantics, $s_0 \in S$. The example uses data points of bit-width four, thus four red solid boxes are mapped for each data point.

Note that fault indices of data point v may differ before and after a program point that reads the data point, even if the value of the data point remains the same. For instance, in Fig. 4a, data point v written at program point p_2 is read at program point p_3 , and new fault indices are assigned to data point v after the data point is live and will be read at program points p_5 and p_6 . Thereby the analysis distinguishes a bit corruption at v^j between program points p_2 and p_3 from the ones between program points p_3 and p_5 or p_6 . No new fault indices are assigned to data point v at program points p_5 and p_6 as the data point is assumed to be killed at those program points.

The fault index coalescing analysis is an iterative process and is performed in two phases: (1) intra-instruction fault index coalescing, where fault indices are coalesced within a program point (Fig. 4b), and (2) inter-instruction fault index coalescing, where fault indices are coalesced across instructions (Fig. 4c). By repeating the two phases, fault index coalescing merges equivalence classes of fault indices backward along the dependency edges, until a fixed point is reached.

Fig. 4b illustrates intra-instruction fault index coalescing of our coalescing example. At program point p_3 , an `and` operation is conducted on argument v and an immediate of bit representation `0001`. According to the intra-instruction fault index coalescing rule of the `and` operation from Algorithm 3, soft errors at fault sites (p_3, v^1) , (p_3, v^2) , and (p_3, v^3) are masked, thus $s((p_3, v^1)) \sim_{R'} s((p_3, v^2)) \sim_{R'} s((p_3, v^3)) \sim_{R'} s_0$, where R' is the temporary equivalence relation in Algorithm 3.

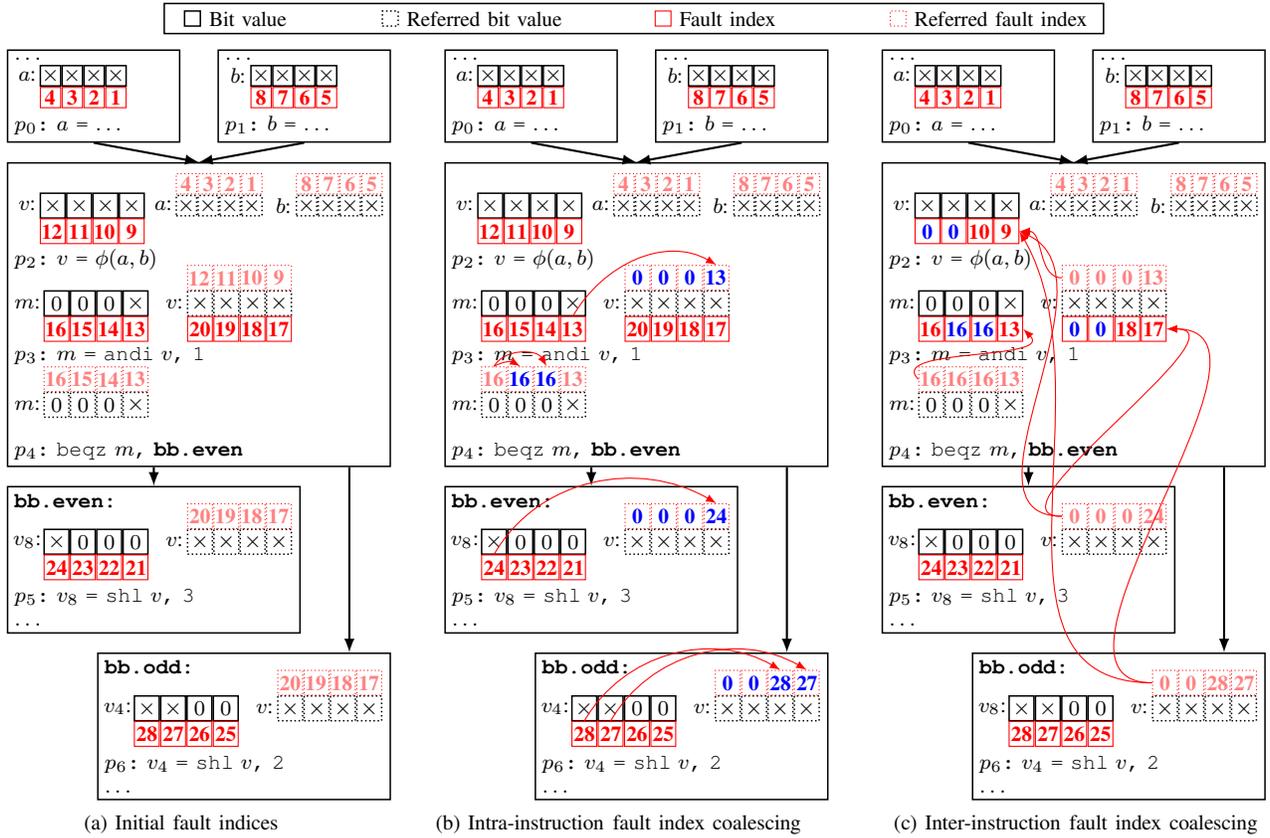


Fig. 4. Iterative fault index coalescing of a fork-after-join CFG snippet using 4-bit data points: (a) initial fault indices are assigned to bits of data points, (b) fault indices are coalesced within their instruction during the intra-instruction fault-index coalescing phase, and (c) fault indices are coalesced across instructions during the inter-instruction fault index coalescing phase. Note that coalescing is a monotonic process that is performed backward along the dependency edges. The example code is in SSA form for brevity, but the proposed method is not limited to SSA form.

Intra-instruction fault index coalescing is performed among the data points read within a program point as well. For instance, in case of the instruction `beqz %mod, bb.even` at program point p_4 , any soft errors occurring at bits where the value is known to be 0 will divert the control flow to `bb.odd`. Thus $s((p_4, m^1)) \sim_{R'} s((p_4, m^2)) \sim_{R'} s((p_4, m^3))$.

Fig. 4c depicts the incorporation of the temporary equivalence relation R' —the result of intra-instruction fault index coalescing—into equivalence relation R by the inter-instruction fault index coalescing analysis. The result of intra-instruction fault index coalescing is applied across instructions only if the new fault indices do not conflict with other access points. For instance, in Fig. 4c, soft errors at data point v after program point p_2 and before program point p_3 affect all three reads of data point v at program points p_3 , p_5 , and p_6 . Hence, $\text{use}(p_2, v) = \{p_3, p_5, p_6\}$. But soft errors between program point p_2 and p_5 or p_6 only affect data point v at program point p_5 or p_6 , therefore, $\text{use}(p_3, v) = \{p_5, p_6\}$. Thus, $[s((p_3, v^j))]_{R'}$, $[s((p_5, v^j))]_{R'}$, and $[s((p_6, v^j))]_{R'}$ are merged into $[s((p_2, v^j))]_R$ only if $s((p_3, v^j)) \sim_{R'} s((p_5, v^j)) \sim_{R'} s((p_6, v^j))$ for any $0 \leq j < 4$. In Fig. 4c, $[s((p_2, v^2))]_R$ and $[s((p_2, v^3))]_R$ are coalesced to $[s_0]_R$ after inter-instruction fault index coalescing, but $[s((p_2, v^0))]_R$ and $[s((p_2, v^1))]_R$ remain the same.

V. VALIDATION

The BEC analysis comprises two monotonic data-flow analyses, the bit-value analysis and the fault index coalescing analysis, and both are the maximal fixed-point (MFP) assignment [23], [25]. In the bit-value analysis, each fault site meets *all* definitions reachable to the fault site under the given CFG and overapproximates its bit-value. The fault index coalescing analysis determines two equivalence classes are equivalent only when it holds for *all* use sites that may follow the fault site under the given CFG. The maximal fixed point guarantees the minimally acceptable soundness criterion of the quality of flow analysis [22], [26].

We implemented the BEC analysis in LLVM 16.0.0 for the RISC-V architecture [9]. To validate the correctness of the implementation, we conducted exhaustive fault injection campaigns on execution traces of programs using an instrumented version of the SPIKE RISC-V ISA simulator [27]. Each fault injection run induces a single soft error at a fault site, which is a particular bit of a register file at a specific clock cycle. For each fault site, individual runs are conducted to probe every bit of the register file exhaustively and one fault injection run is conducted per probed bit. For each fault injection run, the location and the cycle of the fault to be injected are passed as command-line arguments to the ISA

TABLE I
TIME AND DISK SPACE REQUIREMENTS FOR
THE EXHAUSTIVE FAULT INJECTION CAMPAIGN

Benchmark	Time	Disk space
bitcount	0.5 h	1 GB
AES	2 h	7 GB
CRC32	7 h	116 GB
SHA	10 h	100 GB
RSA	50 h	700 GB

TABLE II
CLASSIFICATION OF COMPARISONS

$[s((p, v^i))]_R = [s((q, u^j))]_R \wedge t((p, v^i)) = t((q, u^j))$	Sound, precise
$[s((p, v^i))]_R \neq [s((q, u^j))]_R \wedge t((p, v^i)) = t((q, u^j))$	Sound, imprecise
$[s((p, v^i))]_R = [s((q, u^j))]_R \wedge t((p, v^i)) \neq t((q, u^j))$	Unsound

simulator. The ISA simulator executes the program for each fault injection run from the beginning, flips the to-be-probed bit when the specified execution cycle is reached, and resumes execution.

Corrupted execution traces are generated for each fault injection run and labeled based on their semantics. An execution trace comprises a sequence of executed instructions, side effects caused by the instructions executed such as memory accesses, and observable outcomes of the program.

The validation process highlights several merits of the BEC analysis. The exhaustive fault injection campaign is a highly time- and disk-space consuming process. Table I lists time and disk space required to conduct fault injection campaigns on a *single execution trace* per benchmark, limited to benchmarks where the baseline campaign is tractable. Execution times were obtained on a 3.8 GHz AMD processor and execution traces are generated and classified on the fly and only distinguishable traces are archived to conserve disk space. Besides the time and space costs required for fault injection campaigns, which can easily become infeasible even with small programs, a fault injection campaign is required for each execution trace. Considering that different initial machine states, e.g., due to program input, may generate different execution traces for a program, it is impossible to test all possible execution traces because the input space is generally infinite [28], [29]. In contrast, the BEC analysis needs to run only once per benchmark, at compile-time, and the analysis results hold for all initial machine states and all possible execution traces. The BEC analysis was tractable for all benchmarks, and no significant compile time overhead was observed.

To address the quality of the analysis, let us define $t((p, v^i))$ to represent an execution trace generated from a fault injection run with a fault injected at fault site $(p, v^i) \in F$. Table II lists the three classifications of validation results where $(p, v^i), (q, u^j) \in F$. The BEC analysis is considered sound and precise if it identifies all fault sites that generate identical execution traces when subjected to fault injection. If

the BEC analysis identifies some fault sites as not equivalent but the execution traces are identical, then the BEC analysis is sound but imprecise. This can occur in the presence of dynamic information which is not available at compile time, such as program inputs. We observed a negligible number of sound but imprecise cases, for instance, when analyzing global registers where no assumption is safe, such as `tp`, and `gp` in RISC-V. If two fault injection runs are classified as identical but differ in their execution traces, the analysis is unsound. No such case was observed with the BEC analysis.

VI. EXPERIMENTAL RESULTS

The two use cases of the BEC analysis are evaluated using eight distinctive benchmarks from FISSC [30] and MiBench [31]. We use integer benchmarks for the experiments, but the proposed approach is not limited to specific data types. For instance, x86 AVX registers [32] provide bit-wise and/or operations on IEEE754 floats and can benefit from the proposed method in the same way as integer registers.

This section describes the details of the two use cases and their experimental results.

A. Use Case 1: Fault Injection Campaign Pruning

The BEC analysis classifies the effect of soft errors across all fault sites of a program. Thus, the analysis result can be useful to prune fault injection runs of fault injection campaigns that are known to be masked or identical to the runs that have already been conducted.

Table III shows the number of fault sites that are identified as live and subject to fault injection runs without the BEC analysis (Row “Live in values”) and with the BEC analysis (Row “Live in bits”) on the eight benchmarks. The numbers for “Live in values” are obtained based on the inject-on-read [18]–[20] analysis, which employs value-level analysis to identify fault sites that are live and subject to fault injection runs. Row “Live in bits” shows the numbers of live fault sites that require fault injection runs when analyzed at the granularity of bits by the BEC analysis. Row “Total FI runs pruned” depicts the percentages of fault injection runs pruned by the BEC analysis (rows “Live in bits” versus “Live in values”). Across the eight benchmarks, the BEC analysis pruned up to 30.04% of fault injection runs, at an average of 13.71%.

Rows “Masked bits” and “Inferable bits” in Table III show the breakdown of fault sites pruned by the BEC analysis: Row “Masked bits” depicts the numbers of fault sites pruned because soft errors on those fault sites are analyzed to be masked and dead, and Row “Inferable bits” presents the numbers of fault sites pruned because the effects of the faults are identical to other fault injection runs.

The degree of the effect of the BEC analysis on fault injection pruning varies due to benchmark characteristics. Bit values are rarely known in AES at compile-time. Yet, AES frequently uses `xor` operations to encrypt or decrypt secret keys, which is particularly effective for the BEC analysis. With `xor` operations, fault indices coalesce unconditionally

TABLE III
RESULTS OF FAULT INJECTION PRUNING BY THE PROPOSED STATIC ANALYSIS

	bitcount	dijkstra	CRC32	adpcm enc	adpcm dec	AES	RSA	SHA
Live in values	26 272	230 336	245 760	2 819 904	2 003 744	150 112	1 026 304	421 632
Live in bits	20 571	229 409	211 176	2 424 874	1 653 714	105 025	1 025 436	371 294
Masked bits	2 506	70	7 368	71 000	258 000	680	434	10 660
Inferable bits	3 195	857	27 216	324 030	92 030	44 407	434	39 678
Total FI runs pruned	21.70%	0.40%	14.07%	14.01%	17.47%	30.04%	0.08%	11.94%

TABLE IV
CHANGES IN THE RELIABILITY AGAINST SOFT ERRORS FROM BIT-LEVEL VULNERABILITY-AWARE INSTRUCTION SCHEDULING

	bitcount	dijkstra	CRC32	adpcm enc	adpcm dec	AES	RSA	SHA
Total fault space	541 696	27 286 528	2 922 496	58 426 368	44 085 248	3 180 544	18 295 808	7 483 392
Best reliability	85 018	159 966	348 384	28 401 348	19 400 720	1 928 214	8 650 606	2 559 116
Worst reliability	94 366	166 074	394 040	28 530 244	19 538 104	2 007 194	8 764 640	2 688 188
Worst/Best	111.00%	103.82%	113.11%	100.45%	100.71%	104.10%	101.32%	105.04%
+	+11.00%	+3.82%	+13.11%	+0.45%	+0.71%	+4.10%	+1.32%	+5.04%

across instructions. These characteristics of AES resulted in the highest pruning rate across the set of benchmarks, 30.04%.

BEC achieved a solid 17.47% pruning rate for the ADPCM decoder (adpcm dec), but for a different reason. The decoding process involves an abundant number of bit operations, more importantly, with constant values. This enhances the quality of bit-value analysis, leading to more precise analysis results. The ADPCM encoder and decoder perform internal operations on 4-bit values but clamp to 1-bit or 2-bit values to output. Such characteristics of the benchmarks foster the BEC analysis to identify more fault sites to be masked.

RSA is an adversary case for the BEC analysis because the majority of its operations are arithmetic and thus challenging for bit-value analysis, with noticeable impact on the performance of the fault index coalescing analysis.

B. Use Case 2: Bit-level Vulnerability-aware Instruction Scheduling

Vulnerability-aware instruction scheduling highlights the merit of static analysis. The BEC analysis identifies which fault sites are masked and insensitive to soft errors, and such information can be employed with instruction scheduling to enhance the reliability of programs against soft errors. Instruction scheduling in LLVM is based on list scheduling [33]. First, it maintains a dependency graph among instructions and registers accessed per basic block to examine which instructions in a basic block are ready to be scheduled. If multiple instructions are ready for instruction scheduling, LLVM uses heuristics to choose the one with the highest priority. These heuristics include register pressure, instruction latency, pipeline stalls, etc. The number of fault sites susceptible to soft errors analyzed by the BEC analysis is used as a novel criterion to select the most appropriate instruction to schedule when there is more than one candidate.

Algorithm 4 describes how instruction scheduling is conducted with the BEC analysis as the selection criterion. For each scheduling region, i.e., a basic block, a data dependency graph is created with instructions. The instruction scheduler iterates over the set of instructions until all the instructions

Algorithm 4: Instruction Scheduling for Reliability

Input: a set of instructions P with data dependencies

Output: a sequence of instructions S

- 1 Initialize an empty schedule S .
- 2 **while** $P \neq \emptyset$ **do**
- 3 Choose a set of ready instructions R in P with no unsatisfied data dependencies
- 4 Choose an instruction p in R which kills the most fault sites in bits
- 5 Add the instruction p to S
- 6 Remove the instruction p from P

are scheduled into a list of instructions. When multiple instructions satisfy the data dependency requirement and are ready to be scheduled, the result of the BEC analysis is used as a selection criterion, and the instruction that reduces the most unmasked fault sites is prioritized for scheduling.

Table IV shows the experimental results when the BEC analysis is used as the new selection criterion for instruction scheduling. Row “Total fault space” in Table IV indicates the total number of fault sites per execution trace of benchmarks, and Row “Best reliability” in Table IV shows the numbers of fault sites susceptible to soft errors when the instruction scheduling criterion is set to maximize the number of masked fault sites. Row “Worst reliability” shows the numbers of fault sites susceptible to soft errors when the selection criterion is the opposite. Row “Worst/Best” in Table IV shows the maximum possible reliability improvement against soft errors with the proposed instruction scheduling technique.

The benchmarks that improved noticeably from vulnerability-aware instruction scheduling were CRC32 and bitcount, showing 13.11% and 11.00% reduction in vulnerable fault sites, respectively. Both CRC32 and bitcount have abundant numbers of masked bits, facilitating a high improvement in reliability against soft errors after vulnerability-aware instruction scheduling.

Both the ADPCM encoder and decoder contain large num-

bers of masked bits, but instructions are rather tightly ordered compared to other benchmarks. It restricted scheduling flexibility irrespective of the respective scheduling policy.

AES does not contain many bits masked in live registers. However, the registers stay live relatively long for infrequent read accesses. The effect of masked bits are exacerbated in such long-lived registers even if the masked bits are few. Shortening the live ranges of registers with more live fault sites contributed the most to the improvement of reliability in the case of AES.

RSA lacks masked or inferable bits to be exploited by the new scheduling criterion and it is a highly sequential program with frequent memory accesses. Thus, there was little room for improvement by varying instruction scheduling criteria, yet the proposed instruction scheduling technique achieved a 0.08% improvement.

Bit-level vulnerability-aware instruction scheduling enhanced by the BEC analysis increased the reliability of programs by up to 13.11%, and 4.94% on average. Despite the heuristic nature of instruction scheduling, no degradation of the reliability against soft errors was observed among the benchmarks that we have evaluated. It is worth noting that instruction scheduling does not affect the number of instructions executed per program nor the raw number of fault injection runs required for an exhaustive fault injection campaign, because it does not change the number of data accesses.

VII. RELATED WORK

A. *Static vs. Dynamic Analysis for Reliability*

Several dynamic approaches have been proposed to analyze the propagation of faults at bit-level in hardware components [19], [34] and in execution traces of programs [35]. Dynamic analysis under-approximates program semantics. In contrast, static analysis, which is the proposed method, computes an over-approximation of program semantics. Thus, a single static analysis run covers all possible inputs or execution traces of a program, while dynamic analysis must be performed for every possible input or trace of a program. The essential limitation with dynamic approaches is therefore that it is impossible to test all possible inputs because the input space is generally infinite [28], [29].

Another merit of static analysis is that it can contribute to improving programs fundamentally by optimizing or transforming the program at compile-time based on the analysis results. As a proof of concept, we have conveyed the result of the proposed analysis method to the instruction scheduler of LLVM to enhance the reliability of programs against soft errors.

B. *Fault Injection Pruning*

We have demonstrated the effectiveness of the BEC analysis on fault injection pruning as the first use case. One efficient and widely used fault injection pruning strategy is to conduct fault injection runs just before the value is read. This method was introduced by Smith et al. [36] as a way to determine

equivalent fault classes for permanent and transient faults in values. It was further developed to Inject-on-read [18]–[20], [37]. Unlike BEC, these methods operate at the granularity of values and hence lack optimization opportunities at bit-level.

C. *Instruction Scheduling for Reliability*

As the second use case of the proposed bit-level analysis, we used the analysis results as the criteria for instruction scheduling in LLVM. With the results of the proposed bit-level analysis, instruction scheduling can be determined in a way to maximize the number of fault sites insensitive to soft errors. Rehman et al. [38] proposed reliability-aware instruction scheduling strategies that determine reliability-critical instructions by looking ahead in the instruction sequences. Xu et al. [39] proposed an instruction scheduling strategy that reduces the overall length of the live intervals of registers. These approaches are based on value-level analysis for the criticality or longevity of registers. The proposed bit-level analysis can be readily combined with the previous approaches to further enhance the effectiveness. Instruction scheduling augmented by the BEC analysis enhanced the reliability of programs against soft errors comparable to the improvements achieved by established methods in the field [39]–[41].

VIII. CONCLUSION

We have presented BEC, a static bit-level analysis that enhances the reliability of programs against soft errors, and two of its use cases. The work has been implemented within LLVM 16.0.0 for the RISC-V architecture and validated on an instrumented version of the SPIKE RISC-V ISA simulator using eight benchmarks with distinctive characteristics. The proposed bit-level analysis pruned up to 30.04% of exhaustive fault injection campaigns (13.71% on average), without loss of accuracy. Program vulnerability reduced by up to 13.11% (4.94% on average) through bit-level vulnerability-aware instruction scheduling.

The BEC analysis has been applied to software, but we anticipate this work to be easily extended for hardware testing to reduce production costs. For instance, it can be adopted as a pass of hardware synthesis tools to shorten the chip testing process analytically and systematically.

IX. DATA-AVAILABILITY STATEMENT

The code that supports the findings of this study has been open-sourced on GitHub [10] and archived on Zenodo [42].

ACKNOWLEDGMENT

We thank the anonymous reviewers for their insightful comments and suggestions. This work was partly supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Ministry of Science and ICT (MSIT) (No. 2021-0-00853), and by the BK21 FOUR of the Department of Computer Science and Engineering, Yonsei University, funded by the National Research Foundation of Korea (NRF).

REFERENCES

- [1] T. C. May and M. H. Woods, "Alpha-particle-induced soft errors in dynamic memories," *IEEE Transactions on Electron Devices*, vol. 26, no. 1, pp. 2–9, 1979. doi: [10.1109/T-ED.1979.19370](https://doi.org/10.1109/T-ED.1979.19370)
- [2] R. Baumann, "Soft errors in advanced computer systems," *IEEE Design Test of Computers*, vol. 22, no. 3, pp. 258–266, 2005. doi: [10.1109/MDT.2005.69](https://doi.org/10.1109/MDT.2005.69)
- [3] S. Baffreau, S. Bendhia, M. Ramdani, and E. Sicard, "Characterisation of microcontroller susceptibility to radio frequency interference," in *Proceedings of the Fourth IEEE International Caracas Conference on Devices, Circuits and Systems (Cat. No.02TH8611)*, 2002, pp. I031–I031. doi: [10.1109/ICDCS.2002.1004088](https://doi.org/10.1109/ICDCS.2002.1004088)
- [4] S. Jagannathan, Z. Diggins, N. Mahatme, T. D. Loveless, B. L. Bhuvu, S.-J. Wen, R. Wong, and L. W. Massengill, "Temperature dependence of soft error rate in flip-flop designs," in *2012 IEEE International Reliability Physics Symposium (IRPS)*, 2012, pp. SE.2.1–SE.2.6. doi: [10.1109/IRPS.2012.6241927](https://doi.org/10.1109/IRPS.2012.6241927)
- [5] T. Instruments, "Error Detection in SRAM (Rev. A)," Nov 2020. [Online]. Available: <https://www.ti.com/lit/pdf/spracc0>
- [6] B. Schroeder, E. Pinheiro, and W.-D. Weber, "DRAM errors in the wild: A large-scale field study," *Commun. ACM*, vol. 54, no. 2, p. 100–107, feb 2011. doi: [10.1145/1897816.1897844](https://doi.org/10.1145/1897816.1897844)
- [7] C. Noeldeke, M. Boettcher, U. Mohr, S. Gaisser, M. Alvarez Rua, J. Eickhoff, M. Leslie, M. Von Thun, S. Klinkner, and R. Varatharajoo, "Single event upset investigations on the "flying laptop" satellite mission," *Advances in Space Research*, vol. 67, no. 6, pp. 2000–2009, 2021. doi: [10.1016/j.asr.2020.12.032](https://doi.org/10.1016/j.asr.2020.12.032)
- [8] B. Yuce, P. Schaumont, and M. Witteman, "Fault attacks on secure embedded software: Threats, design, and evaluation," *Journal of Hardware and Systems Security*, vol. 2, no. 2, pp. 111–130, Jun 2018. doi: [10.1007/s41635-018-0038-1](https://doi.org/10.1007/s41635-018-0038-1)
- [9] A. Waterman and K. Asanovic, "The RISC-V Instruction Set Manual, Volume I: User-Level ISA," <https://five-embeddev.com/riscv-isa-manual/latest/riscv-spec.html>, March 2019.
- [10] Y. Ko, "BEC analysis GitHub repository," <https://github.com/yousunko/BEC>, 2023.
- [11] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006.
- [12] S. S. Muchnick, *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998.
- [13] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4, p. 451–490, Oct 1991. doi: [10.1145/115372.115320](https://doi.org/10.1145/115372.115320)
- [14] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '77. New York, NY, USA: Association for Computing Machinery, 1977, p. 238–252. doi: [10.1145/512950.512973](https://doi.org/10.1145/512950.512973)
- [15] S. McCanne and V. Jacobson, "The BSD packet filter: A new architecture for user-level packet capture," in *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, ser. USENIX'93. USA: USENIX Association, 1993, p. 2.
- [16] A. Dixit, R. Heald, and A. Wood, "Trends from ten years of soft error experimentation," *Proc. 2009 IEEE Workshop on Silicon Errors in Logic-System Effects (SELSE)*, March, 2009. [Online]. Available: <https://cir.nii.ac.jp/crid/1574231875991217152>
- [17] V. Sridharan, J. Stearley, N. DeBardleben, S. Blanchard, and S. Gurumurthi, "Feng Shui of supercomputer memory: Positional effects in DRAM and SRAM faults," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. New York, NY, USA: Association for Computing Machinery, 2013. doi: [10.1145/2503210.2503257](https://doi.org/10.1145/2503210.2503257)
- [18] H. Schirmeier, C. Borchert, and O. Spinczyk, "Avoiding pitfalls in fault-injection based comparison of program susceptibility to soft errors," in *Proceedings of the 2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, ser. DSN '15. USA: IEEE Computer Society, 2015, p. 319–330. doi: [10.1109/DSN.2015.44](https://doi.org/10.1109/DSN.2015.44)
- [19] L. Berrojo, I. González, F. Corno, M. Sonza Reorda, G. Squillero, L. Entrena, and C. López, "New techniques for speeding-up fault-injection campaigns," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '02. USA: IEEE Computer Society, 2002, p. 847. doi: [10.1109/DATE.2002.998398](https://doi.org/10.1109/DATE.2002.998398)
- [20] B. Sangchoolie, F. Ayatollahi, R. Johansson, and J. Karlsson, "A comparison of inject-on-read and inject-on-write in ISA-level fault injection," in *Proceedings of the 2015 11th European Dependable Computing Conference (EDCC)*, ser. EDCC '15. USA: IEEE Computer Society, 2015, p. 178–189. doi: [10.1109/EDCC.2015.24](https://doi.org/10.1109/EDCC.2015.24)
- [21] L. C. Paulson, "Defining functions on equivalence classes," *ACM Trans. Comput. Logic*, vol. 7, no. 4, p. 658–675, oct 2006. doi: [10.1145/1183278.1183280](https://doi.org/10.1145/1183278.1183280)
- [22] M. N. Wegman and F. K. Zadeck, "Constant propagation with conditional branches," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 2, p. 181–210, Apr 1991. doi: [10.1145/103135.103136](https://doi.org/10.1145/103135.103136)
- [23] U. Khedker, A. Sanyal, and B. Karkare, *Data Flow Analysis: Theory and Practice*, 1st ed. USA: CRC Press, Inc., 2009.
- [24] O. Ore, "Theory of equivalence relations," *Duke Math. J.*, vol. 9, no. 1, pp. 573–627, 1942. [Online]. Available: <http://dml.mathdoc.fr/item/1077493381>
- [25] J. B. Kam and J. D. Ullman, "Monotone data flow analysis frameworks," *Acta Inf.*, vol. 7, no. 3, p. 305–317, sep 1977. doi: [10.1007/BF00290339](https://doi.org/10.1007/BF00290339)
- [26] S. L. Graham and M. Wegman, "A fast and usually linear algorithm for global flow analysis," in *Proceedings of the 2nd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '75. New York, NY, USA: Association for Computing Machinery, 1975, p. 22–34. doi: [10.1145/512976.512979](https://doi.org/10.1145/512976.512979)
- [27] RISC-V International, "Spike RISC-V ISA simulator GitHub repository," <https://github.com/riscv/riscv-isa-sim>, 2019.
- [28] M. Dalla Preda, R. Giacobazzi, and N. Marastoni, "Formal framework for reasoning about the precision of dynamic analysis," in *Static Analysis: 27th International Symposium, SAS 2020, Virtual Event, November 18–20, 2020, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2020, p. 178–199. doi: [10.1007/978-3-030-65474-0_9](https://doi.org/10.1007/978-3-030-65474-0_9)
- [29] M. D. Preda, "Towards a unifying framework for tuning analysis precision by program transformation," in *Recent Developments in the Design and Implementation of Programming Languages*, ser. OpenAccess Series in Informatics (OASiCS), F. S. de Boer and J. Mauro, Eds., vol. 86. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, pp. 4:1–4:22. doi: [10.4230/OASiCS.Gabrielli.4](https://doi.org/10.4230/OASiCS.Gabrielli.4)
- [30] L. Dureuil, G. Petiot, M.-L. Potet, T.-H. Le, A. Crohen, and P. de Choudens, "FISSC: A fault injection and simulation secure collection," in *Proceedings of the 35th International Conference on Computer Safety, Reliability, and Security, SAFECOMP 2016*, A. Skavhaug, J. Guiochet, and F. Bitsch, Eds. Cham: Springer International Publishing, 2016, pp. 3–11. doi: [10.1007/978-3-319-45477-1_1](https://doi.org/10.1007/978-3-319-45477-1_1)
- [31] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, ser. WWC '01. USA: IEEE Computer Society, 2001, p. 3–14. doi: [10.1109/WWC.2001.990739](https://doi.org/10.1109/WWC.2001.990739)
- [32] Intel Corporation, *Intel 64 and IA-32 Architectures Software Developer's Manual*, June 2023. [Online]. Available: <https://cdrdv2.intel.com/v1/dl/getContent/671436>
- [33] J. Schutten, "List scheduling revisited," *Operations Research Letters*, vol. 18, no. 4, pp. 167–170, 1996. doi: [10.1016/0167-6377\(95\)00057-7](https://doi.org/10.1016/0167-6377(95)00057-7)
- [34] C. Dietrich, A. Schmider, O. Pusz, G. P. Vayá, and D. Lohmann, "Cross-layer fault-space pruning for hardware-assisted fault injection," in *Proceedings of the 55th Annual Design Automation Conference*, ser. DAC '18. New York, NY, USA: Association for Computing Machinery, 2018. doi: [10.1145/3195970.3196019](https://doi.org/10.1145/3195970.3196019)
- [35] O. Pusz, C. Dietrich, and D. Lohmann, "Data-flow-sensitive fault-space pruning for the injection of transient hardware faults," in *Proceedings of the 22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, ser. LCTES 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 97–109. doi: [10.1145/3461648.3463851](https://doi.org/10.1145/3461648.3463851)
- [36] D. Smith, B. Johnson, J. Profeta, and D. Bozzolo, "A method to determine equivalent fault classes for permanent and transient faults," in *Annual Reliability and Maintainability Symposium 1995 Proceedings*, 1995, pp. 418–424. doi: [10.1109/RAMS.1995.513278](https://doi.org/10.1109/RAMS.1995.513278)

- [37] Y. Ko, A. Bradbury, B. Burgstaller, and R. Mullins, "Trace-and-brace (TAB): Bespoke software countermeasures against soft errors," in *Proceedings of the 23rd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, ser. LCTES 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 73–85. doi: [10.1145/3519941.3535070](https://doi.org/10.1145/3519941.3535070)
- [38] S. Rehman, M. Shafique, and J. Henkel, "Instruction scheduling for reliability-aware compilation," in *Proceedings of the 49th Annual Design Automation Conference*, ser. DAC '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 1292–1300. doi: [10.1145/2228360.2228601](https://doi.org/10.1145/2228360.2228601)
- [39] J. Xu, Q. Tan, and H. Zhou, "Scheduling instructions for soft errors in register files," in *Proceedings of the 2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing*, ser. DASC '11. USA: IEEE Computer Society, 2011, p. 305–312. doi: [10.1109/DASC.2011.69](https://doi.org/10.1109/DASC.2011.69)
- [40] J. Yan and W. Zhang, "Compiler-guided register reliability improvement against soft errors," in *Proceedings of the 5th ACM International Conference on Embedded Software*, ser. EMSOFT '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 203–209. doi: [10.1145/1086228.1086266](https://doi.org/10.1145/1086228.1086266)
- [41] J. Xu, Q. Tan, and R. Shen, "The instruction scheduling for soft errors based on data flow analysis," in *Proceedings of the 2009 15th IEEE Pacific Rim International Symposium on Dependable Computing*, ser. PRDC '09. USA: IEEE Computer Society, 2009, p. 372–378. doi: [10.1109/PRDC.2009.65](https://doi.org/10.1109/PRDC.2009.65)
- [42] Y. Ko, "BEC analysis Zenodo archive," 2023. doi: [10.5281/zenodo.10317898](https://doi.org/10.5281/zenodo.10317898)