Investigating the Impact of High-Level Software Design on Low-Level Hardware Fault Resilience

Bohan Zhang[1], Lishan Yang[2], Guanpeng Li[1], Hui Xu[3]
[1] Computer Science Department, University of Iowa, Iowa City, IA, USA
[2] Department of Computer Science, George Mason University, Fairfax, VA, USA
[3] School of Computer Science, Fudan University, Shanghai, China
bohan-zhang-1@uiowa.edu, lyang28@gmu.edu, guanpeng-@uiowa.edu xuh@fudan.edu.cn

***Abstract*—Silent Data Corruptions (SDCs) have become an insurmountable issue that threatens the system reliability. General strategies for protecting programs from SDCs, such as dual modular redundancy, incur intolerable overheads. Another strategy is Algorithm-Based Fault Tolerance which is highly bounded to the specific algorithm and hard to generalize. In this study, we find different implementations of the same algorithm may lead to very different SDC probabilities. We conduct a characterization study to quantify the differences and investigate the root causes. The insights we derive could help and guide the developers in software engineering domain to design programs that is naturally resilient.**

***Index Terms*—Silent Data Corruption, Error Resilience, Fault Injection, SDC, Program Analysis, Software Testing**

I. Introduction

With the fast development of semiconductor technology, the number of transistors is significantly increasing in computer systems, leading to a huge gain of computing power. However, the risk of soft errors (i.e., hardware transient faults) [1] is increasing due to various factors such as silicon decay or cosmic radiation, which can lead to data loss or corruption. In fields that require precise computation results, such as medical applications and numerical computing, soft errors can cause catastrophic events such as critical safety issues and economic losses. Therefore, reliability analysis and enhancement are necessary to protect systems from soft errors.

Silent Data Corruptions (SDCs) have become an insurmountable issue threatening system reliability [2]–[5]. It is difficult to detect and diagnose compared to the other error types, i.e., crashes and hangs. When an SDC happens in program execution, it does not have noticeable signs and symptoms but omits incorrect outputs. Typically, SDC probability is measured through fault injection experiments. A higher SDC probability shows that the application is more vulnerable. To mitigate the SDC, an industry solution usually applies Triple Modular Redundancy (TMR) to the vulnerable instructions of the program that we need to protect [6]–[8] but carrying out such a protection strategy usually requires colossal energy and computation resources. For example, applying full protection using TMR on a program will give rise to 100% overhead.

Blindly applying full protection introduces huge performance degradation. In this work, for the first time, we bring the idea of studying coding behavior from the software engineering domain to reliability by investigating *the impact of different implementations on application resilience*. Different programmers have their own coding styles and preferences, resulting in various differences at the code level (i.e., function calls and instruction orders), even when implementing the same algorithm. We collect five different implementations of BubbleSort from GitHub and conduct fault injection experiments for each implementation to obtain the overall SDC probabilities. Comparing these SDC probabilities, we find that some implementations exhibit low SDC probability and high performance. To find out the reasons behind, we propose a methodology to evaluate every difference in each two of those implementations and why these differences lead to different SDC probabilities. Insights are derived as the results to guide programmers or software engineers to write optimized reliable programs.

Our contributions in this paper are summarized as follows:

- We conduct a detailed characterization study on five different implementations of BubbleSort and show that different implementations have different overall SDC probabilities.
- We perform the per-instruction fault injection to every instruction of every implementation in BubbleSort to evaluate the SDC probabilities of each instruction.
- We propose an approach to systematically analyze and quantify the differences of implementations and their impact on the SDC probability.
- Based on the observations from the characterization study, insights of improving reliability can be derived. We showcase this by identifying an observed difference and derive the insight accordingly.

The remaining of the paper is organized as follows. In In Section II, we describe our approach. Then, we demonstrate results and Observations in Section III. In Section IV and V, we present the future work and conclusion.

II. Methodology

A. Fault Model

In our research, we utilize a prevalent single-bit fault model in the field of error resilience [9]–[12]. Our study is specifically directed towards soft errors in the computing units of processors such as pipeline stages, flip-flops, and functional units. We omit the memory or cache faults, as we presume

that the Error Correction Code (ECC) technique can safeguard them. Faults in the control flow logic of processors, such as illegal addresses, are not taken into account since they can be easily detected. However, the program may execute to a wrong but legal branch as we consider the program-level control flow in our fault model.

B. LLFI

In this work, we perform the fault injection campaign using LLFI (Low-Level Fault Injection) which is a tool for evaluating the robustness of software systems against hardware faults [13], [14]. LLFI injects faults into compiled binary code at the low-level intermediate representation (LLVM-IR) level [15], [16], which allows researchers and developers to study the impact of faults on program behaviors. LLFI provides a low-cost, flexible and customizable platform for experimenting with different fault scenarios and evaluating the resilience of a system against these faults.

C. Our Approach

Figure 1 shows the high-level workflow of our approach. The key of this approach is to identify *observed differences (ODs)* between implementations of the same algorithm and investigate why these ODs lead to different SDC probabilities. First, given a target algorithm, we collect implementations from GitHub or other similar sources. Note that the correctness of implementations is verified in this step. For every implementation, we use the same randomly generated input with a reasonable large input size to provide adequate dynamic instructions (DI) for accurate fault injection results. For each implementation, we perform 1,000 random fault injection experiments to get the overall SDC probability. For each static instruction, we retrieve the DI count by profiling and use it as a proxy to reflect the application performance (i.e., more DIs lead to longer execution time). We also conduct a per-instruction fault injection campaign for each of the implementations to obtain the SDC probability of each static instruction.

To identify an OD, for the same algorithm, we first divide the algorithm into standard components that each implementation must follow. For each component of different implementations, the comparison is done at the source code level, then IR level. The comparison at the source code level excludes the differences in variable names and format variations such as indent and space, then any other differences are marked as an identified difference and we further look into the IR level. At the IR level, the differences include (1) the types of instructions, (2) the number of instructions, and (3) the order of instructions. If an identified difference has been confirmed at the two levels (both source code and IR levels), then we mark it as an OD.

In addition, the SDC probabilities of each component is calculated using the SDC probability of each static instruction in this component weighted by the DI count. The equation is shown as following:

$$SDC\_Prob_{Component} = \sum_{i=1}^{n} \frac{SDC\_Prob_i \times \#DI_i}{\#DI_{overall}}, \quad (1)$$

where n denotes the number of static instructions in the component.

After we collect all the ODs, we perform a *controlled experiment* for those two implementations. The primary purpose of the controlled experiments is to verify whether those ODs affect the SDC probabilities. Figure 2 shows the logic of the controlled experiment. For each OD, we compare the SDC probability of the component in two implementations. Our hypothesis is that if the SDC probability of a component is higher than the other one, replacing the component with the lower SDC probability would result in a modified implementation with a lower overall SDC probability. To confirm this hypothesis, we perform another fault injection campaign to the modified implementation. We compare the overall SDC probability of the modified implementation with the original implementation. If the modified implementation has lower SDC probability and the DI number is similar (i.e., similar performance), then the result has verified the hypothesis and an insight is generated. These insights could guide the software engineers to write programs with lower SDC probabilities.

III. Preliminary Results

A. Experimental Setup

We use a Debian server with two 20-core Intel CPU processors to run fault injection experiments in parallel. We start over with a simple sorting algorithm: BubbleSort. Algorithm 1 shows the pseudo code. It repeatedly steps through the input list element by element, comparing the current element with the one next to it, swapping their values if smaller. We divide the algorithm into three components: Outer Loop, Inner Loop, and Swaption. we crawl five different C++ implementations from different authors from GitHub, noted as P1 – P5. we randomly generate an array of 20,000 integers ranging from 0 to 1,000,000 as the program input for all implementations.

Algorithm 1 Bubble Sort

```
1:  procedure BUBBLESORT(A)
2:      n ← |A|
3:      for i ← 1 to n − 1 do                    ▷ Outer loop
4:          for j ← 0 to n − i − 1 do            ▷ Inner loop
5:              if A_j > A_{j+1} then            ▷ Swaption
6:                  swap A_j and A_{j+1}
7:              end if
8:          end for
9:      end for
10: end procedure
```

B. Overall Analysis of Implementations

For each implementation, we perform a fault injection campaign by randomly sampling 1000 fault sites (i.e., fault

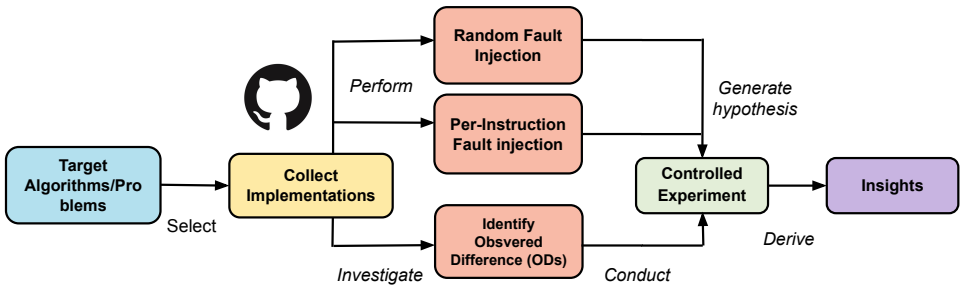

Fig. 1. Overview of our approach.

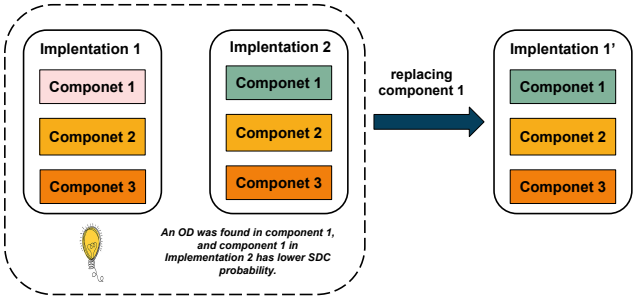

Fig. 2. Overview of a controlled experiment.

locations) and injecting one fault per program execution to obtain the SDC probability. Our fault injection measurement yields an error bar from 0.26% to 3.10% for the 95% confidence intervals.

Figure 4 shows the obtained SDC probabilities of the five implementations. The SDC probabilities vary from 4% (P5) to 26.7% (P2), which shows that different high-level code designs have different SDC probabilities. Table I presents the DI count of these five implementations, ranging from 9,088,698,022 (P5) to 5,489,077,998 (P4). Note that the DI count of P5 is significantly higher than the other four implementations due to the poor implementation in P5. For the other four implementations, the inner loop terminates when its index is equal to or greater than the size of the input array (i.e., the loop iterations start from 0 till $n-i-1$). In P5, the inner loop iterates from 0 to $n-1$, causing many duplicated loop cycles. Many SDCs are masked during loop cycles but introduce huge overhead.

C. Observed Difference and Insight

In this subsection, we present an OD to showcase our methodology. The presented OD is the swaption of P1 and P2:

Observed Difference: The swaption in P1 is through a function call whereas it is directly inlined in P2.

In detail, the swaption in P1 uses the built-in function from the C++ code libraries and P2 implements the swaption functionality inside the inner loop. Considering the LLVM-IR level code difference, because P2 uses the inline function, it must use more instructions than P1, such as *getelementptr* and *sext*. These redundant instructions contributes to a higher SDC probability.

TABLE I
NUMBER OF DYNAMIC INSTRUCTIONS OF 5 PROGRAMS.

Implementations	Number of Dynamic Instructions
P1	5,889,058,005
P2	6,087,856,752
P3	5,685,594,374
P4	5,489,077,998
P5	9,088,698,022

We conduct per-instruction fault injection campaigns for each implementation by performing 100 random fault injection experiments to each static instruction, resulting in an error bar from 0.00% to 3.10% for the 95% confidence intervals. From the per-instruction fault injection results, Figure 3 shows the SDC probability of the swaption component for the P1 and P2, which are 22.53% and 40.81%, respectively.

The DI counts of P1 and P2 are very close, indicating that the performance is also similar. Since the SDC probability of the swaption in P1 is lower than that of P2, if we replace the swaption in P1 with the implementation of P1 to generate *P2'*, then the overall SDC probability of P2' will be lower than P2. To confirm this hypothesis, we perform another set of fault injection experiments to evaluate the overall SDC probabilities of P1, P2, and P2', as shown in Figure 5. The SDC probability of P2' is only 18.5%, which is reduced by 8.2% compared to P2. Hence, the result proves our hypothesis. Therefore, we derive an insight:

Insight: Using function call other than inlined function could reduce the SDC probabilities.

We can apply this insight to any other high-level code design to write a robust program with lower SDC probability. We emphasize that the purpose of this analysis is to *derive useful insights* that could guide programmers to write more reliable applications rather than just improving the resilience of P2.

IV. FUTURE WORK

In this section, we outline the possible future directions.

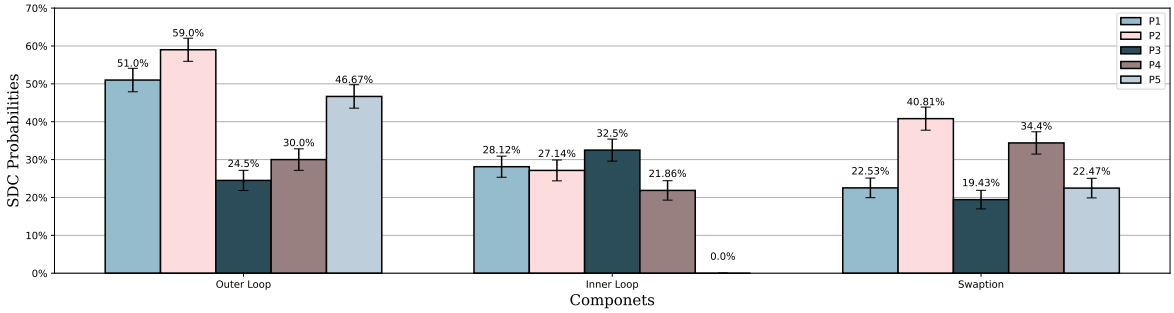

Fig. 3. SDC Probabilities for each component in each implementation

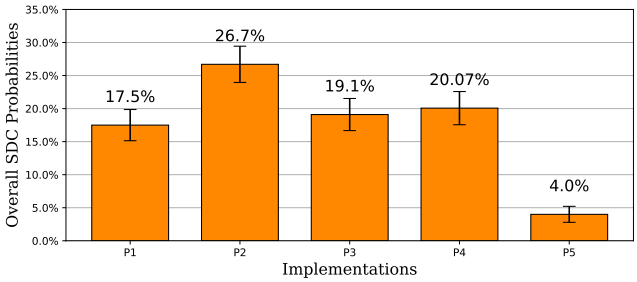

Fig. 4. Overall SDC Probabilities of 5 Programs

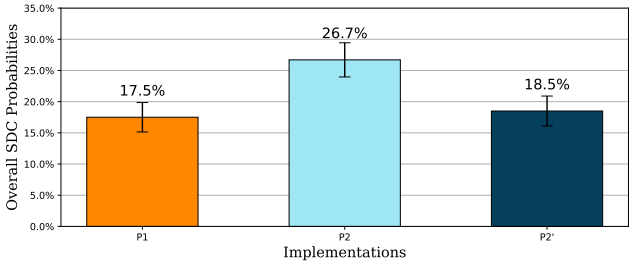

Fig. 5. Overall SDC Probabilities with modified versions

A. Understanding High-Level Software Design on Low-Level Hardware Fault Resilience

One of the main future directions we are planning is to have a more comprehensive study on characterizing how different code fragment designs will impact on hardware fault resilience. We plan a large-scale experiment which includes a broad spectrum of benchmark programs from various domains, quantifying their design and resilience. Furthermore, we are interested in understanding the root-causes why certain code designs lead to high or low hardware fault resilience, which may offer insights in generating new code structures of high reliability on our own. Our preliminary study on the applications of BubbleSort in this paper provides a positive feedback on the feasibility of the direction.

B. Guidance on Implementing Reliable Software

Based on the characterization, we are planning a further in-depth study on analyzing the implications of the observations, generating a set of useful guidance which provide a mean of designing reliable programs for software developers. Without time-consuming fault injections or additional analysis, developers can choose the alternative code fragments and designs of high reliability, improving the natural fault tolerance of the program out of the box.

V. Conclusion

In this work, we focus on analyzing the resilience of different implementations given a target algorithm/problem to help software engineers write more reliable programs. We propose a methodology to collect different implementations, perform reliability analysis, identify observed differences and conduct a controlled experiment on the ODs, to derive useful insights. We present a complete analysis on BubbleSort to showcase the feasibility of this approach. In the future, we aim to generalizing the methodology and coming up with a complete guidance for writing more reliable code especially for safety critical applications [17]–[20].

References

[1] K. Lee, A. Shrivastava, I. Issenin, N. Dutt, and N. Venkatasubramanian, "Mitigating soft error failures for multimedia applications by selective data protection," in *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, 2006, pp. 411–420.

[2] H. D. Dixit, S. Pendharkar, M. Beadon, C. Mason, T. Chakravarthy, B. Muthiah, and S. Sankar, "Silent data corruptions at scale," *arXiv preprint arXiv:2102.11245*, 2021.

[3] 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.

[4] L. Yang, B. Nie, A. Jog, and E. Smirni, "Sugar: Speeding up gpgpu application resilience estimation with input sizing," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 5, no. 1, pp. 1–29, 2021.

[5] D. Kuvaiskii, R. Faqeh, P. Bhatotia, P. Felber, and C. Fetzer, "Haft: Hardware-assisted fault tolerance," in *Proceedings of the Eleventh European Conference on Computer Systems*, 2016, pp. 1–17.

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

[7] 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.

[8] D. Kuvaiskii, O. Oleksenko, P. Bhatotia, P. Felber, and C. Fetzer, "Elzar: Triple modular redundancy using intel advanced vector extensions," *Technical Report*, 2016.

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

[10] S. Feng, S. Gupta, A. Ansari, and S. Mahlke, "Shoestring: probabilistic soft error reliability on the cheap," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 1, pp. 385–396, 2010.

[11] 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.

[12] M. H. Rahman, A. Shamji, S. Guo, and G. Li, "Peppa-x: finding program test inputs to bound silent data corruption vulnerability in hpc applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–13.

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

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

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

[16] G. Li, K. Pattabiraman, C.-Y. Cher, and P. Bose, "Understanding error propagation in gpgpu applications," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2016, pp. 240–251.

[17] 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)*, 2022, pp. 85–94.

[18] J. C. Knight, "Safety critical systems: challenges and directions," in *Proceedings of the 24th international conference on software engineering*, 2002, pp. 547–550.

[19] G. Li, Y. Li, S. Jha, T. Tsai, M. Sullivan, S. K. S. Hari, Z. Kalbarczyk, and R. Iyer, "Av-fuzzer: Finding safety violations in autonomous driving systems," in *2020 IEEE 31st international symposium on software reliability engineering (ISSRE)*. IEEE, 2020, pp. 25–36.

[20] M. Gharib and P. Giorgini, "Modeling and analyzing information integrity in safety critical systems," in *Advanced Information Systems Engineering Workshops: CAiSE 2013 International Workshops, Valencia, Spain, June 17-21, 2013. Proceedings 25*. Springer, 2013, pp. 524–529.