Instruction-level Reliability Improvement for Embedded Systems

Hakan Tekgul
Computer Engineering Department
Georgia Institute of Technology
Atlanta, Georgia, USA
0002-5704-8190

Ozcan Ozturk
Department of Computer Engineering
Bilkent University
Ankara, Turkey
ozturk@cs.bilkent.edu.tr

***Abstract*—With the increasing number of applications in embedded computing systems, it became indispensable for the system designers to consider multiple objectives including power, performance, and reliability. Among these, reliability is a bigger constraint for safety critical applications. For example, fault tolerance of transportation systems has become very critical with the use of many embedded on-board devices. There are many techniques proposed in the past decade to increase the fault tolerance of such systems. However, many of these techniques come with a significant overhead, which make them infeasible in most of the embedded execution scenarios. Motivated by this observation, our main contribution in this paper is to propose and evaluate an instruction criticality based reliable source code generation algorithm. Specifically, we propose an instruction ranking formula based on our detailed fault injection experiments. We use instruction rankings along with the overhead tolerance limits and generate a source code with increased fault tolerance. The primary goal behind this work is to improve reliability of an application while keeping the performance effects minimal. We apply state-of-the-art reliability techniques to evaluate our approach on a set of benchmarks. Our experimental results show that, the proposed approach achieves up to 8% decrease in error rates with only 10% performance overhead. The error rates further decrease with higher overhead tolerances.**

***Index Terms*—Fault Tolerance, Reliability, Instruction Criticality, Embedded Systems**

I. Introduction

Reliability of embedded systems is a major concern these days, especially in the context of performance-hungry environments. The design and creation of these embedded computing systems have significant design restraints in terms of execution time, power and cost. These real-time computing constraints not only pose a threat for emerging generations, but also indicate criticality in many application domains such as transportation, aviation, and space. Since the number of applications, functionality of programs, and the amount of data they process tend to increase everyday, creating systems with minimum error rates have become even more difficult.

This work has been supported in part by a grant from Turkish Academy of Sciences.

978-1-7281-5429-9/20/$31.00 ©2020 IEEE

One important aspect of any computing system is its processor and peripheral devices. The variety of processors increase everyday with various levels of fault tolerance. Since the processor is responsible for processing low-level instructions, its reliability should be considered with utmost importance. When an instruction is processed, registers' roles are critical for the output of the program. For any critical control system, a random fault in the processor, datapath or ALU can cause catastrophic failures [1]. Therefore, the role of processors for fault-tolerant systems and instruction reliability is highly significant.

Hence, we try to improve processor reliability in this work. Specifically, by considering the executing instructions and registers, we present an approach to increase reliability on embedded systems. We focus on low-level instructions of a program and estimate their tolerance for errors. We use these tolerance values to sort instructions according to their criticality for correct execution. Then, starting with the least fault tolerant instruction, we apply the state-of-the-art techniques on instructions and generate a reliable low-level source code.

Our main goal in this study is to propose a technique that quantifies *instruction criticality* and reduce the overhead of current reliability techniques based on fault-injection experiments and control flow graph (CFG) analysis. As a result, our approach would be used in accordance with the current reliability techniques to generate *reliable source code* based on a predetermined overhead limit. We use the term *instruction criticality* to rank different instructions based on how dependable they should be. In deciding this, there are various factors to be considered from loop count to instruction type. By measuring instruction criticality using these factors, selective instruction protection can be applied where the overhead limit of the system would decide on the critical instructions to be executed in a more dependable way. In our approach, we use bit-flip injections to each instruction in the control flow graph (CFG) and analyze the rate of Silent Data Corruption (SDC) as well as the amount of Crash. By comparing these rates in different data dependent source codes and CFGs, we propose two special metrics for instruction criticality based on instruction type. These metrics are dependent on crashing and data corruption of a program where a detailed explanation is provided in Section 4.

We conduct fault-injection experiments using LLFI [2], an LLVM [3] based fault injector. More specifically, we inject bit-flip faults on low-level intermediate source code representation (IR). After analyzing effects on criticality, we propose an algorithm that would generate dependable source code. That is, our approach generates an optimized source code based on the overhead limit and number of critical instructions. This algorithm can further be used to generate dependable code for embedded systems with a performance, power, or other limitations. Furthermore, it can be applied to environments where Crash or SDC is not tolerable at all.

Our experimental results on our benchmarks indicate that the proposed reliable code generation algorithm and instruction criticality formula can increase fault tolerance of the system while reducing the associated overhead. For a dynamic overhead limit of 10% it was observed that our algorithm can increase fault tolerance up to 8%, while for an overhead limit of 70%, our approach increased the fault tolerance by 30-40%.

In the next section, we present the details of our proposed formula and code generation algorithm. In Section III, we give an experimental evaluation of the algorithm and code generation where we compare our results with different benchmarks. The paper is concluded in Section IV with a summary of our major observations and possible future research directions.

II. Proposed Approach

A. Problem Definition and High-Level View

Our goal in this paper is to present and evaluate a formula to quantify instruction criticality and sort low-level instructions by their fault tolerance. This instruction criticality metric would be used to generate reliable source code and reduce the overhead in system reliability approaches applied today.

Our proposed approach for reliable code generation takes three inputs: Original source code to be improved upon, an overhead limit for critical systems and a variable α that decides on the significance of SDC and Crash tolerance of the system. The main objective behind the approach is to decide on the number of instructions that can be improved based on our proposed criticality formula and then generate reliable source code. Hence, our approach outputs a very similar source code as the input but with added fault tolerance. Note that, reliability technique applied is orthogonal to our approach. We focus on where and when to apply this reliability technique. Therefore, our proposed mechanism can potentially work with any source code level reliability technique.

In Figure 1, we present a high-level description of our approach for reliable code generation and instruction criticality. After taking in the inputs, we calculate each instruction's criticality value based on their location in the program and their place on our definition of SDC and Crash metric. Finally, this criticality value is improved by taking α and the number of times the instruction would dynamically execute into consideration. Dynamic execution statistics are collected using profilers embedded to the compiler framework. After each instruction's criticality is calculated, we sort these instructions by their criticality and eliminate the instructions that are outside the given overhead limit. Therefore, we only apply the reliability enhancement technique to instructions within the tolerated overhead limit. While our baseline overhead limitation considers performance as a percentage, it can be extended to other types of overheads such as energy. In the next step, we apply the state-of-the-art reliability techniques on the instructions to be improved upon and then output the reliable source code.

Note that our approach may not generate a program that is fully fault-tolerant. Our goal in this work is to show that instruction criticality can actually be quantified as a value and then can be further used for reliable code generation in a selective manner. We achieve this by defining two metrics, SDC and Crash rankings, based on instruction type. These metrics are created and defined after conducting thousands of fault-injection experiments and analyzing the data. Our definition of these metrics is important for our work and affects the instruction criticality value significantly. The details of the fault-injection experiments and quantifying instruction criticality are explained in the next subsections.

B. Instruction Criticality

In order to design a strong and widely applicable instruction ranking that would work for any source code with any amount of data, fault-injection tests are critical. The rates of SDC and Crash and their location in the control flow exhibit certain patterns. Since we observed that Silent Data Corruption rates are much higher in the final instructions of a program, there must be a direct relation between Silent Data Corruption and instruction location in the control flow. On the other hand, crash rates are much higher in the first basic blocks of CFG, which indicates an inversely proportional relationship between instruction location and Crash rates.

Furthermore, our fault-injection experiments also showed that specific instruction types are less effective in Silent Data Corruption or Crash of a program. By observing average SDC and Crash values of each instruction type in various applications, we sorted the different instruction types separately for Silent Data Corruption and Crash of a program. Hence, we define two seperate reliability metrics for Silent Data Corruption and Crash.

We define two variables, namely SDC_m and C_m, where SDC_m index has the value of the current instruction type from the SDC metric and C_m index has the value of the current instruction type in control flow from Crash metric, respectively. Finally, we define the variable $ILCF$ to represent the instruction location in the control flow in a normalized form. Specifically, $ILCF$ (Instruction Location in the Control Flow) can be calculated by dividing the index of the current instruction to the total number of instructions in the program.

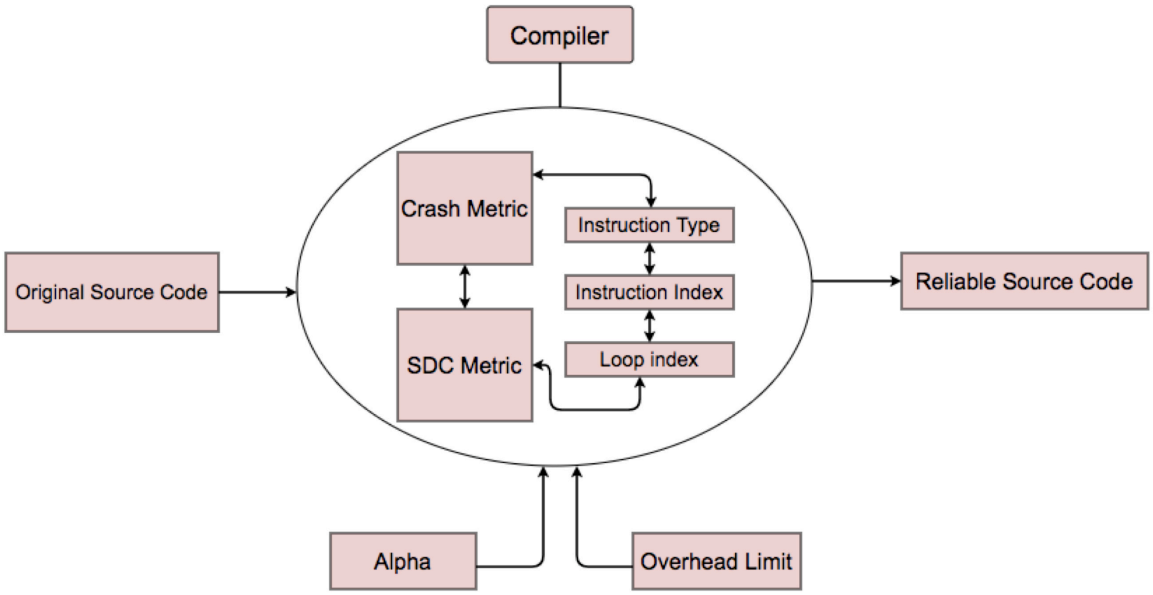

Fig. 1. High level sketch of our reliable source code generation approach based on instruction criticality formula. α is defined as a user input where it quantifies the relative importance of SDC and Crash tolerance of the original source code. The overhead limit is another user input that sets the limit for the instructions to be improved in the source code.

We capture these properties in our instruction criticality (IC) formulation as follows:

$$IC = (SDC_m \times ILCF) + (\frac{1}{C_m \times ILCF}). \quad (1)$$

The formula above calculates the criticality of any instruction in any source code. By looking at the instruction type of the current instruction from the metrics defined above and by calculating the right instruction location, we capture both SDC and Crash significance of an instruction. Note that the SDC and Crash rankings are created solely from the fault injection experiments on LLFI. It is important to state that these metrics could actually be parameterized, depending on the application. Since we conducted the fault injection experiments with a wide range of applications, we created our own rankings for SDC and Crash. We use these metrics to calculate the instruction criticality in our approach. However, depending on the application, there could be changes in the rankings of instruction types, which can be used for an application specific fault reduction technique. Our goal in this work is to create a common framework that can be used in any application.

As stated before, we introduce a variable α to the instruction criticality formula so that the user can decide whether the reliable source code should be more tolerant towards SDC or Crash. This variable will be an input to our approach and will be used in our formula to decrease or increase the criticality of SDC or Crash values.

Finally, based on the minimum and maximum values of each set of terms in our instruction criticality expression, we normalize the values to better reflect their impact. Our overall IC rank calculation is done according to the below formulation:

$$IC = (\alpha)(SDCm \times ILCF) + (1-\alpha)(\frac{1}{Cm \times ILCF}) \quad (2)$$

C. Reliable Code Generation

Reliable code generation depends on the instruction criticality formula we presented. By using the instruction criticality and the input source code, we apply the formula to each instruction in the code. First, we parse the IR file using built-in LLVM [3] libraries and generate an instruction index for each instruction. We save each instruction's index and the instruction type which then is used with SDC metric and Crash metric to calculate instruction's criticality. Then, we sort these instructions based on their criticality value. According to the most critical instructions and the given overhead limit, we modify the source code. This overhead tolerance limit is considered as the percentage increase in execution cycles.

As stated before, the reliability technique to be used in this setup is orthogonal to our approach. Even though we used EDDI (Error Detection by Duplicating Instructions) [4] for experimental results, any other source code reliability technique could be used. Since the aim of this paper is to capture the importance of instruction criticality and reduce the total overhead, different reliability techniques can potentially make use of our approach.

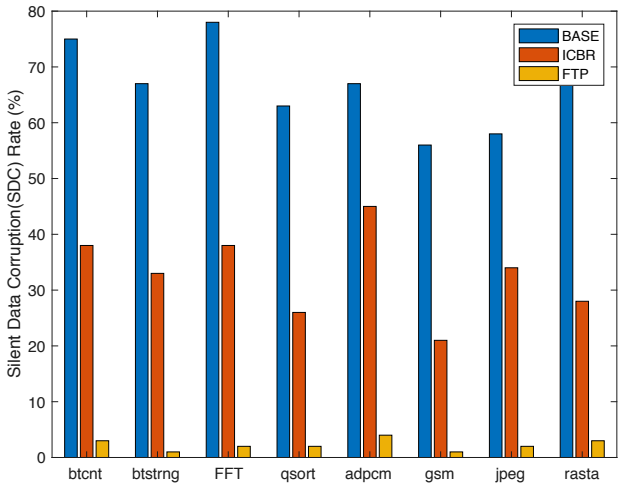

Fig. 2. Fault injection results for BASE, ICBR, and FTP for our benchmarks.

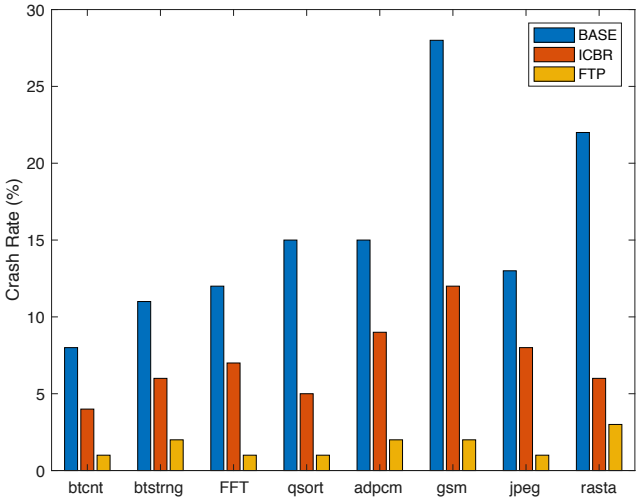

Fig. 3. Fault injection results for BASE, ICBR, and FTP for our benchmarks.

III. Experimental Evaluation

We tested our reliable code generation algorithm on different benchmarks from MediaBench [5] and MiBench [6]. The set of benchmark codes used in our experiments are given in Table 1. The third column of this table explains the functionality implemented by each benchmark. The next two columns give the number of basic blocks and code size in kilobytes, respectively. The last column gives the dynamic number of instructions executed.

We collected statistics for a number of different applications in each benchmark and compared the SDC and Crash rates with non-modified source codes. After implementing our code generation algorithm, we again used LLFI [2] on different benchmark applications where we first injected faults in a random manner without any modification. Then, for different overhead limits and different α values, we injected faults similarly to our proposed approach. In order to get accurate results, a thousand fault injections were conducted on the source code using a random number generator. All experiments are repeated five times and the average values of those experiments were reported.

For each benchmark code in our experimental suite, we performed experiments with 3 different versions, and Table 2 lists the base simulation parameters used in our experiments. Unless stated otherwise, our results are collected using these parameters. We use α as 0.5 for the default value to keep the significance of SDC and Crash the same. Note that this value can easily be changed by the user. We also set the overhead limit as 70% to compare and prove the usefulness of our formula.

Experimental results presented in this paper are based on two inputs to our approach; α and the overhead limit. As stated before, based on these inputs, we collected results on 3 different versions of each source code; BASE, ICBR, and FTP. It is significant to state that both MiBench and MediaBench produced similar results.

Our first set of results describe the fault injections on our benchmarks for BASE, ICBR, and FTP to analyze the SDC rate. As can be seen from Figure 2, data corruption rates decrease significantly with our approach compared to the BASE case. More specifically, our approach reduces SDC rate from 67% to 32% on average when compared with the BASE case. On the other hand, our results are higher when compared to the average SDC rate of 8% for FTP. However, this is expected since our approach limits the performance overhead to 70% by default, whereas FTP incurs 115% of performance overhead on the average as a result of full protection.

The next set of experiments show the effects of fault injections on Crash rates. As can be seen from Figure 3, BASE results with an average of 8% to 28% Crash rates, whereas this range is reduced to an average of 1% to 5% for FTP. Our approach, on the other hand, have Crash rates ranging from 4% to 14%. On the average, our approach reduces the Crash rates from 17% to 8% when compared to BASE. Similar to SDC, Crash rates are also higher with respect to FTP due to the performance overhead limitation enforced.

Based on the results shown in Figures 2 and 3, one can observe that data corruption rates and crashes are reduced with a limited overhead.

IV. Conclusion

In this paper, we attempt to decrease the overhead caused by state of the art reliability techniques by presenting an instruction criticality formula and a reliable code generation algorithm. We show that instruction criticality is heavily dependent on instruction type, instruction location in control flow, and execution frequency. Taking these into consideration, we present our approach on reliable code generation where current reliability techniques are applied only to most critical instructions. Our LLVM-based implementation provides encouraging results in our experiments on MiBench and MediaBench. We observe 35% decrease in Silent Data Corruption

TABLE I
BENCHMARKS USED IN OUR EXPERIMENTS AND THEIR CHARACTERISTICS.

Benchmark	Source	Type	Number of Basic Blocks	Code Size (KB)	Instruction Count (mil)
btcnt	MiBench [6]	Automative	138	98	688.3
btstrng	MiBench [6]	Automative	56	48.9	327.3
FFT	MiBench [6]	Telecomm	44	69.2	238.89
qsort	MiBench [6]	Automative	78	72.3	513.8
adpcm	MediaBench [5]	Compression	22	8	1.2
gsm	MediaBench [5]	Telecomm	98	438	7.09
jpeg	MediaBench [5]	Decompression	112	488.8	18.65
rasta	MediaBench [5]	Feature Extraction	189	269	24.86

TABLE II
BASELINE PARAMETERS USED IN OUR EXPERIMENTS.

Parameter	**Default Value**
α	0.50
Overhead limit	70%

(SDC) rate and a 10% decrease in Crash rate on average when we limit the performance by 70%. It is also important to note that, even with a small overhead as low as 10%, we are able to increase fault tolerance up to 8%.

REFERENCES

[1] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. U. Jain, and M. Stumm, "Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, 2014, pp. 249–265. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/yuan

[2] 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*, Aug 2015, pp. 11–16.

[3] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, ser. CGO '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 75–. [Online]. Available: http://dl.acm.org/citation.cfm?id=977395.977673

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

[5] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "Mediabench: a tool for evaluating and synthesizing multimedia and communications systems," in *Proceedings of 30th Annual International Symposium on Microarchitecture*, Dec 1997, pp. 330–335.

[6] 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)*, Dec 2001, pp. 3–14.