Imrpoved and Efficient Music Genre Classification

Hakan Tekgul
University of Illinois Urbana-Champaign
Urbana, Illinois
tekgul2@illinois.edu

Raimi Shah
University of Illinois Urbana-Champaign
Urbana, Illinois
rsshah2@illinois.edu

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.

CCS CONCEPTS

• **Computer systems organization → Embedded systems**; **Reliability**;

KEYWORDS

Fault Tolerance, Reliability, Instruction Criticality, Embedded Systems

ACM Reference Format:
Hakan Tekgul and Raimi Shah. 2018. Imrpoved and Efficient Music Genre Classification. In *Proceedings of Embedded Systems Week Conference (ESWEEK 18').* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 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. Because 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. To deal with this situation and the competitive nature of computing market, various techniques have been developed to mask negligible errors and increase the overall system performance. Unfortunately, most of these techniques have been proven to increase execution time, cost and energy consumption in many applications. For example, in control systems, current reliability techniques such as triple modular redundancy (TMR) [14] cause a significant overhead in the system and may damage the overall system performance and affect the mission-control. Therefore, new reliability techniques that can minimize the overheads while satisfying the contstraints must be considered.

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 [32]. 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

ESWEEK 18', October 2018, Italy

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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.

In addition to having different metrics for Crash and SDC, we also focus on other aspects of instruction criticality. The different properties that would affect criticality of an instruction are based on fault injection results and pattern analysis. First, we enumerate instructions based on their type. Then, we also consider the instruction location on the control flow to determine its effect on the rest of the program. With weighing in all these factors, we propose a formula to measure instruction criticality for any source code. Based on this formula, we apply our reliability improvement techniques on the most critical instructions. We expect this approach to significantly reduce the overhead caused by blindly applying reliability techniques.

We conduct fault-injection experiments using LLFI [13], an LLVM [10] 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. To summarize, we make three main contributions in this paper:

- We present an instruction criticality formula dependent on various factors in the program. This formula has two main metrics, SDC and Crash. These metrics are based on instruction type and instruction location in control flow. When calculating the criticality, we take loop count and data-dependency into consideration and add those factors in our formulation as well.
- We implement an algorithm based on the presented formula where our approach would transform a source code into a dependable one. This source code can be further used in embedded systems with low tolerance to soft errors.
- We report experimental data that describe the overall effectiveness of our proposed dependable code generation algorithm.

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 compare our work to other instruction criticality quantification approaches and reliable code generation algorithms. In section 3, we provide a brief background on instruction criticality, current reliability techniques and fault-tolerant systems. Section 4 presents the details of our proposed formula and code generation algorithm. In Section 5, 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 6 with a summary of our major observations and possible future research directions.

2 BACKGROUND

There are many reliability techniques to increase fault-tolerance on real-time computing systems. From register files to compiler optimizations, different approaches at different layers were proposed in the last few decades. In terms of both hardware and software, these error mitigation techniques tend to be inadequate for big amounts of data due to their overheads. The standard techniques implemented in the literature are based upon fault injection simulations [30] or mathematical models [18] developed to estimate circuit-level soft error rates.

In terms of circuit-level reliability techniques, a model such as Soft Error Monte-Carlo Modeling estimates and checks for soft error propagation across multiple gates in a combinatorial circuit [18]. Furthermore, for architecture-level techniques, Architecturally Correct Execution (ACE) analysis is performed to estimate the vulnerability of a processor component [17]. However, since ACE analysis ignores error masking, it is only suitable for regular structures such as cache or register files. As for the software-level error redundancy techniques, fault-injection methodologies are quite crucial. Current software level approaches such as Error Detection using Duplicated Instructions (EDDI) [20] and Software Implemented Fault Tolerance (SWIFT) [26] increase system reliability by copying current instructions in the control flow and placing comparison instructions before any branch or data related instruction. Other than that, Simultaneous Redundant Threading (SRT) is used as a tolerance technique for transient fault coverage [25]. Rebaudengo et al proposes a soft-error detection approach by automatically introducing data and code redundancy into an existing program written using a high-level language [21]. This approach frees the programmer from the cost and responsibility of introducing suitable Error Detecting Mechanisms (EDMs) in its code. Another traditional concurrent error detection approach is to use hardware redundancy and N version programming [9], [1]. Unfortunately, as stated, these kinds of techniques result with significant performance overheads [7].

Other than the reliability techniques, there are many reliability estimation models and reliability metrics developed to mitigate and mask soft errors in a program [2], [16], [28]. Even though the proposed metrics are very useful to measure the vulnerability of a system, they may not be sufficient for low-level analysis and error estimation. Therefore, quantifying software reliability remains as an important problem in software development.

In order to come up with an accurate model to increase fault-tolerance in a computing system, data dependency and Crash rates must be considered. Data dependency is especially significant since any corruption in data could lead to catastrophic results. To deal with error propagation due to data dependency, Silent Data Corruption (SDC) is defined. Any computer component that deals with

any amount of data is subject to SDC. Fiala et al discusses the detection and correction of Silent Data Corruption in high-performance computing by providing a transparent redundancy [5]. When an undetected data corruption occurs in a system, it may result in cascading failures, in which the system may run for a period of time with undetected initial error causing increasingly more problems until it is ultimately detected. In order to deal with SDC, Ni et al proposes an automatic software-based approach that protects applications from SDC by leveraging compile-time analysis and program markup [19]. Other than that, watchdog processors are used for concurrent system-level error detection, which requires less hardware than replication [15]. Moreover, Crash rates in a program are also very significant. A single bit flip in the program could make the whole system crash and lead to system failures. Some of the computing environments such as safety critical systems are required to be crash tolerant as they can not risk the possibility of a system failure. In his paper, Chakravorty proposes a fast fault recovery protocol to implement a crash tolerant computing system by using the idea of processor virtualization [3]. Therefore, data dependency with SDC and Crash rates must be analyzed to create a fault-tolerant embedded system.

3 RELATED WORK

Instruction criticality was researched by several studies where the focus was generally on a shared memory multiprocessor [12] or on a cross-layer approach [23]. Tune et al presented an approach on Quantifying Instruction Criticality that predicts critical paths in a program and defines tautness as a measure of importance of critical instructions [31]. Fields have proposed a directed acyclic graph (DAG) model for characterizing program microexecutions on uniprocessors [6]. Under such a model, critical path analysis can be applied and instructions' slack values can be used to quantify instruction criticality. Lebeck extends the uniprocessor DAG model to characterize parallel program executions on shared memory multiprocessor systems [12]. Semeen proposed a similar model for Reliability-Driven Selective Instruction Protection where instruction-level Error Masking Index (IMI) and Error Propagation Index (EPI) are defined to create a Reliability Profit Function for instruction criticality [23]. Finally, Subramaniam et. al propose several processor enhancement techniques that can exploit criticality information of an instruction with an implementation of a small criticality predictor [29].

Furthermore, reliable code generation was also the focus of many research projects. After his Selective Instruction Protection, Semeen proposed a multi-layer approach to achieve reliable code generation and execution at compilation and system software layers for embedded systems by building a schedule table offline to optimize the Reliability-Timing penalty [24]. Besides these, Semeen also proposed a compilation tehcnique for reliability-aware software transformations where spatial and temporal vulnerability is considered [22]. Schwarz et al, on the other hand, presented a reliable software development methodology for safety related applications for reliable source code dependent on a simulation [27]. Finally, Excoffon et al discussed adaptive fault tolerance mechanisms (FTMs) and provided an estimation model for the quantification of system's resilience or reliability where the impact of assumptions of FTM selection was thoroughly analyzed [4].

As compared to these works, our approach considers factors such as instruction location, loop count or instruction type to create a more efficient and fast algorithm for dependable code generation and instruction criticality. Specifically, compared to the Selective Instruction Protection work of Semeen [23], our approach focuses on each instruction's characteristics such as its location or type and considers the repeated usage in a loop. Furthermore, even though the Reliability Profit Function Semeen defined in his work is similar to our instruction criticality formula, we try to consider both data corruption and crash rates and give user the ability to concentrate more on one of the error types than the other. Our work not only provides flexibility to the user but also considers any instruction-level error that might be encountered in an embedded system.

4 PROPOSED APPROACH

4.1 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

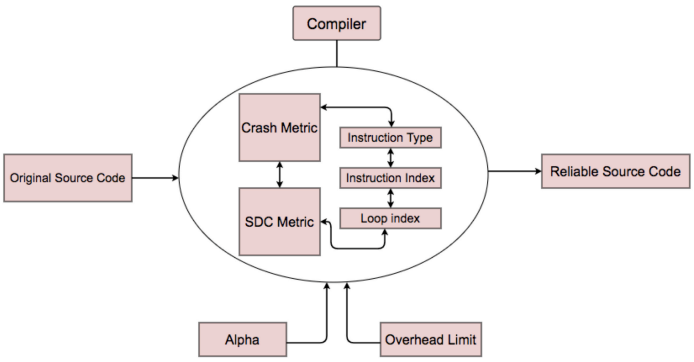

Figure 1: High level sketch of our reliable source code generation approach based on instruction criticality formula. α is defined as a user input where it quantifies the relative importance of SDC and Crash tolerance of the original source code. The overhead limit is another user input that sets the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

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.

4.2 Fault-Injection

Fault-injection experiments and their results were used to estimate a criticality value for each instruction in the source code. These experiments were conducted for each instruction in a program considering the program's Control Flow Graph (CFG). Based on the Crash and Silent Data Corruption rates in each basic block, different patterns were extracted to measure criticality based on CFG.

The experiments were conducted using LLFI [13], which injects faults into the LLVM IR of the application source code. Since LLFI injects faults systematically in a reproducible manner, bit-flip injections were conducted on each instruction for at least a thousand times. Specifically, on each fault injection, a single bit is flipped on a certain instructions in a random manner. Then, for each instruction, we trace the effects of fault injection on the application code and output. We also look at the affected instructions after each injection to spot instruction types that may damage the output of the program the most.

We used different types of programs from different application domains. From programs that perform many floating operations (Factorial) to programs that are mostly dependent on huge amounts of data (QSort), we injected faults to a wide variety of applications. To have sufficient coverage, we tested our approach on programs that had very different and specific control flows with nested if statements and for loops (Nested If/Else). We tried to analyze how a change in control flow would affect the results of a program. Hence, we were able to deduce a pattern on instruction criticality that can be applied on any application.

Our pattern analysis was based on the SDC and Crash rates of a program where we calculated the average fault percentage of each instruction according to its location. Based on this analysis, we observed that Silent Data Corruption (SDC) rates tend to increase if the instruction is in the later basic blocks of the program, as seen in Figure 2. We also see that crash rates increase if the instruction is located at the beginning of the program, as shown in Figure 3. Based on these observations, we conclude that location of the instruction in the control flow is a very significant aspect in the fault analysis. After pattern analysis, we compared the fault injection results for each instruction type. By taking an average of the Crash and SDC rates for each instruction type, we developed two separate metrics to describe SDC tolerance and Crash tolerance for any source code. These two metrics and instruction location pattern are the main factors in our criticality decisions. Based on our analyses, we observe that some of the LLVM IR instructions such as *allocate* or *getelementptr* were the least tolerant for Crash, whereas other LLVM IR instructions such as *load, add or mul* were the least tolerant for Silent Data Corruption.

In Figures 2 and 3, the results from the fault-injection experiments and how they are related to the instruction location in the control flow is presented. For each instruction in the control flow, an instruction index is defined from 0 to the last available instruction's index. Then, the SDC and Crash values are presented in the figures where each instruction's error rate can be mapped to its index in the control flow. Note that the figures are created by using linear regression on each data set where the curves indicate the proportional relationship between instruction index and different error rates. These results are significant for our approach since some error types tend to occur later in the program whereas some other error types occur in the beginning of the program. Hence, Figures 2 and 3 not only present our motivation behind the instruction criticality formula but also shows the significance of instruction location in the control flow. Details of the SDC and Crash rankings and the proposed formula for instruction criticality will be presented in the next subsection with other factors that affect instruction criticality.

4.3 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, as seen in Figures 2 and 3.

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

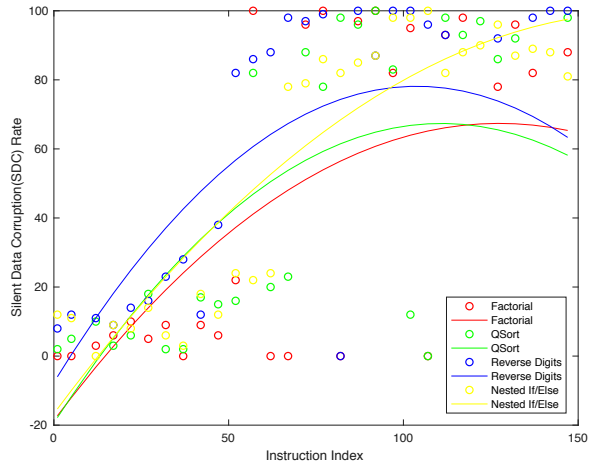

Figure 2: Relationship between the instruction index in control flow and its Silent Data Corruption percentage for different applications (Factorial, QSort, Reverse Digits, Nested If/Else).

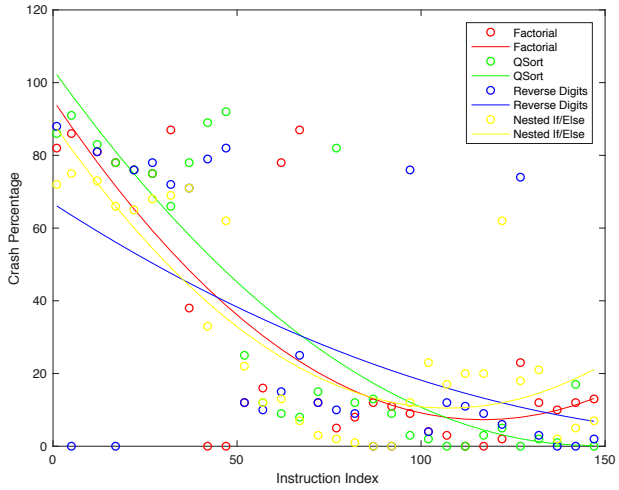

Figure 3: Relationship between the instruction index in control flow and its Crash rates for different applications (Factorial, QSort, Reverse Digits, Nested If/Else).

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.

The motivation behind these metrics is shown in Figures 4 and 5. These figures show significant results regarding the role of instruction type in our instruction criticality formula. As can be seen from these figures, some instruction types tend to crash or corrupt data significantly more than others. This, in turn, affects the corresponding instructions's vulnerability. Our fault injection results show that *allocate* or *load* operations are significant for both SDC and Crash. Moreover, since computations can affect the rate of Silent

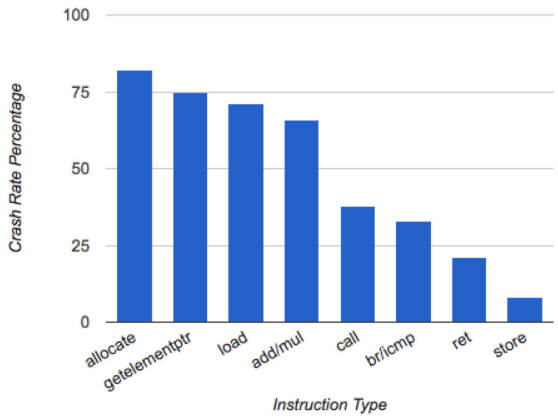

Figure 4: Crash rates for the most critical instructions. Rates are given as a percentage of their effect in the result.

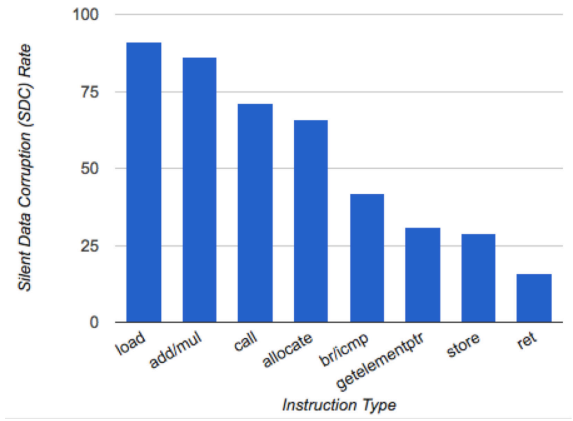

Figure 5: SDC rates for the most critical instructions. Rates are given as a percentage of their effect in the result.

Data Corruption, it is observed that *add* or *mul* instructions are not SDC tolerant. By using the results based on instruction types, we ranked the instructions as shown in Table 1. Note that, for any other instruction that is not listed, the least-tolerant instruction value is used when calculating the instruction criticality of a specific instruction. While instruction types given in Table 1 are LLVM IR specific, our approach can easily be adopted to any low-level source code. More specifically, similar error injections can be applied to obtain instruction type ranking specific to the underlying architecture or instruction set. As can be seen from this table, most of these instructions are commonly found in any instruction set. Therefore, altough we use an LLVM specific setup, our approach is applicable to any architecture.

Based on the SDC/Crash values with instruction location, we can use the rankings defined in Table 1 to create the first part of our instruction criticality formula. We define two variables, namely SDCm and Cm, where SDCm index has the value of the current

Table 1: Individual instruction type rankings are presented where the first instruction type is the most significant type for Silent Data Corruption (or crash). The ranking values next to the instruction types should be considered for computing the instruction criticality. Instruction types that are not listed are treated as the least significant ranking.

Crash Rank	SDC Rank
1. *allocate*	8. *load*
2. *getelementptr*	7. *add/mul*
3. *load*	6. *call*
4. *add/mul*	5. *allocate*
5. *call*	4. *br/icmp*
6. *br/icmp*	3. *getelementptr*
7. *ret*	2. *store*
8. *store*	1. *ret*

instruction type from the SDC metric and Cm 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. We capture these properties in our instruction criticality (IC) formulation as follows:

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

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 defined in Table 1 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.

In addition to these, there are other factors that directly affect instruction criticality. While SDC or Crash of a program is highly significant for reliability, repeated usage of a single instruction will increase its potential damage. If an instruction is repeated constantly in a program (such as in a loop), probability of a soft error happening on that instruction would be higher. If a bit is flipped in such an instruction, the repeated use of that damaged instruction may produce catastrophic results. Based on this observation, we take loop index or loop count of an instruction into consideration where this loop count can be obtained from the compiler or a profiler.

Furthermore, although the value of SDC captures the level of data dependency of a program to a certain extent, other factors for data dependency must also be considered. In order to better estimate the effects of data dependency, we define Data Dependency Constant (DDC) in our instruction criticality expression. Based on our experiments, we observe that bottom-up dataflow analysis fail to include major data dependency effects. Specifically, only a few instruction types are critically affected with huge amounts of data. Therefore, we use the DDC multiplier to increase the criticality of memory instructions (such as *allocate*, *load*, or *store*). We apply DDC only to memory instructions since the data processing of a program are mainly dependent on these instructions. This way, our instruction criticality formula also considers data dependency of a given program.

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:

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

In the above expression, LoopCount indicates the number of iterations a specific instruction gets executed as part of a loop (in case of a single execution this term is assumed to be 0). Our algorithm to calculate this IC metric and respective reliable source code generation will be presented in the next subsection.

4.4 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 [10] 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. After this point, we have the most critical instructions sorted which we modify the source code based on the given overhead limit. This overhead tolerance limit is considered as the percentage increase in execution cycles.

Specifically, we apply one of the state-of-the-art reliability techniques to the most critical number of instructions in such a way that the overhead limit will be met. After applying these, we generate reliable source code with minimal overhead. The algorithm to achieve this code generation is presented in Algorithm 1. Note that the input variable overhead in the algorithm is to be captured dynamically depending on the dynamic execution time of the program. It is fair to state at this point that ignoring least-critical instructions when applying reliability improvements increases performance and reduces other potential overheads.

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) [20] 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.

```
inputs : IR_file, α, Overhead
output : Reliable Source Code
Instruction_Types[] ⟵ ParseIRFile(IR_file)
size ⟵ size(Instruction_Types)
for i ← 0 to size do
    for j ← 0 to 8 do
        if SDCm[j] = Instruction_Types[i] then
            current_SDC ⟵ SDCm[i]
        end
        if Cm[j] = Instruction_Types[i] then
            current_Crash ⟵ Cm[i]
        end
    end
    ILCF ⟵ i/size
    if Instruction_Types[i] =' allocate'||'load'||'store'
     then
        DDC ⟵ 0.55
    end
    else
        DDC ⟵ 0
    end
    IC_Array[i][Instruction_Types[i]] ⟵ alpha ∗
     current_SDC ∗ ILCF + (1 − alpha)(1/(current_Crash ∗
     ILCF) + DDC + LLVM.LoopCount
end
MergeSort(IC_Array)
index ⟵ 0
while dynamic_execution ≤ Overhead do
    ReliableCode ⟵ Apply_Technique(index, IR_file)
    index ⟵ index + 1
end
return ReliableCode
```

Algorithm 1: Our algorithm to generate reliable source code based on our instruction criticality formula.

5 EXPERIMENTAL EVALUATION

5.1 Setup

We tested our reliable code generation algorithm on different benchmarks from Media Bench [11] and MiBench [8]. The set of benchmark codes used in our experiments are given in Table 2. 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 nymber 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 [13] 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, at least a thousand fault injections were conducted on the source code using a random number generator. All experiments are repeated multiple times and the average values of those experiments were reported.

Table 2: Benchmarks used in our experiments and their characteristics.

Benchmark	Source	Type	Number of Basic Blocks	Code Size (KB)	Instr Count (mil)
btcnt	MiBench [8]	Automative	138	98	688.3
btstrng	MiBench [8]	Automative	56	48.9	327.3
FFT	MiBench [8]	Telecomm	44	69.2	238.89
qsort	MiBench [8]	Automative	78	72.3	513.8
adpcm	MediaBench [11]	Compression	22	8	1.2
gsm	MediaBench [11]	Telecomm	98	438	7.09
jpeg	MediaBench [11]	Decompression	112	488.8	18.65
rasta	MediaBench [11]	Feature Extraction	189	269	24.86

Table 3: Baseline parameters used in our experiments.

Parameter	Default Value
α	0.50
Overhead limit	70%
DDC	0.55

For each benchmark code in our experimental suite, we performed experiments with 3 different versions, which can be summarized as follows.
1)*BASE*: The base execution does not employ any optimization where we injected randomized faults into unmodified LLVM IR code and collected data.
2)*ICBR*: This is our instruction criticality based reliability (ICBR) enhancement approach where we applied our reliable code generation algorithm and collected both SDC and Crash results.
3)*FTP*: Finally, we conducted fault injection experiments on fully tolerant and protected (FTP) programs so that we could see how close we are to a fully fault-tolerant system with a given α and overhead.

Finally, Table 3 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 since we want 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. Finally, we use 0.55 as the default value for DDC after our analysis and normalization of our IC formula.

5.2 Results

Experimental results presented in this paper are based on two inputs to our approach; α and the overhead limit. As stated before, based

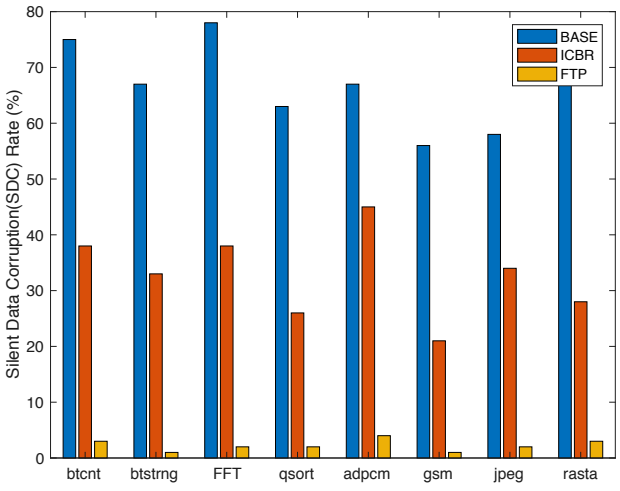

Figure 6: Fault injection results for BASE, ICBR and FTP for our benchmarks.

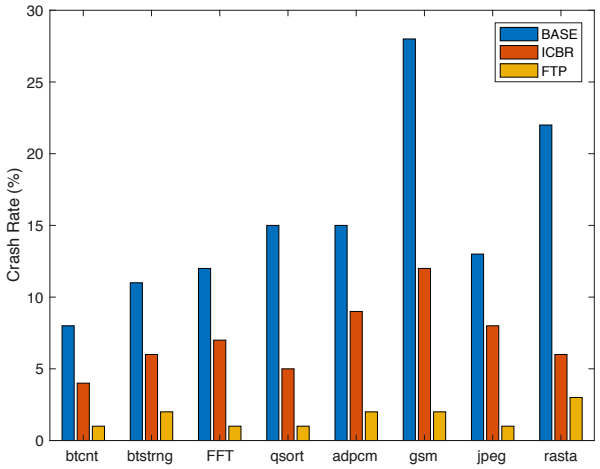

Figure 7: Fault injection results for BASE, ICBR and FTP for our benchmarks.

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 BASE, ICBR and FTP to analyze the SDC rate. Based on analysis with our criticality formula, we set α to 0.5 and the overhead limit to 70%. As stated before, these parameters are chosen this way to compare our approach with different versions in a detailed way. As can be seen from Figure 6, data corruption rates decrease significantly with our approach compared to the BASE case. More specficically, 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

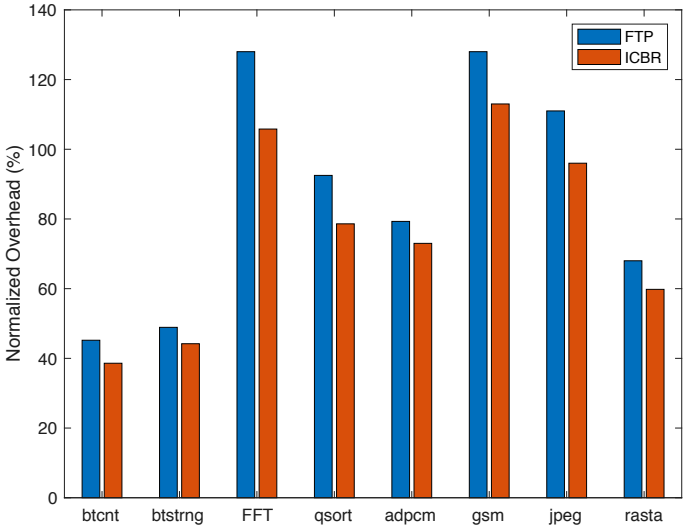

Figure 8: Performance overhead in FTP and ICBR normalized with respect to BASE.

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 7, 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 Figure 6 and 7, one can observe that data corruption rates and crashes are reduced with a limited overhead.

In the next set of experiments, we present the performance overhead of FTP and ICBR results where the execution time is normalized with respect to BASE. As can be seen in Figure 8, the two overhead results are similar where ICBR has shorter execution time compared to FTP. More specifically, FTP results indicate an average of 101% overhead with respect to BASE whereas ICBR data indicate an average of 83% overhead with respect to BASE.

5.3 Sensitivity Analysis

As explained before, α is used to specify the significance of the data corruption or Crash tolerance in an application. Hence, we change α for each source code and analyze the sensitivity of the variable in the error rates. The α parameter is significant for our results since in some applications the Crash rates are considerably low.

Figures 9 and 10 shows the SDC and Crash rates with increasing α values for our benchmarks. We kept the overhead limit constant at 70% to only focus on the effect of α in our approach. Starting from 0, we increased α until 1 with 0.25 increments. As seen from Figure 9, the rate of SDC decreases for each benchmark with increasing

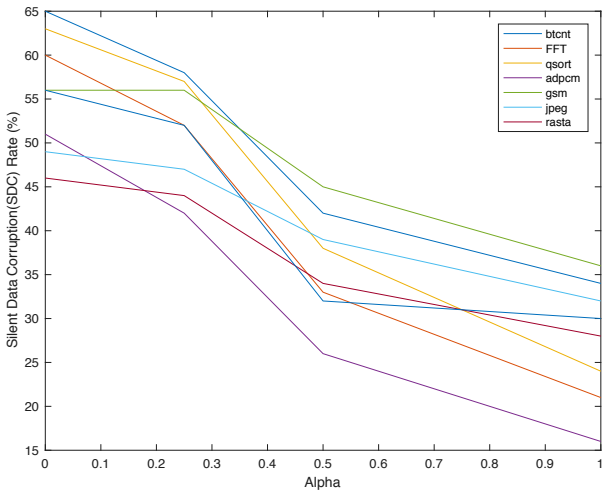

Figure 9: Sensitivity of Silent Data Corruption (SDC) on α for our benchmarks.

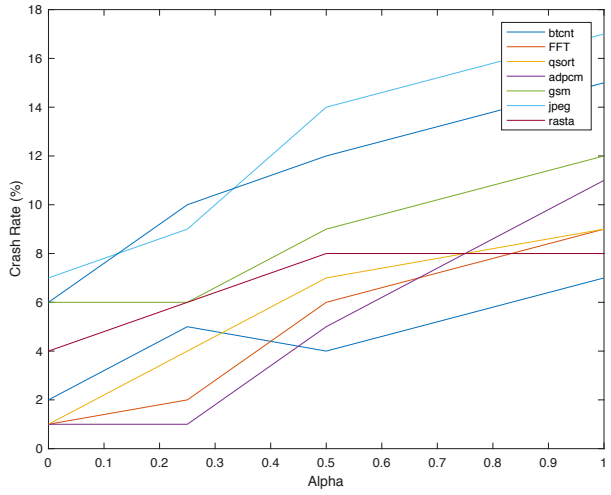

Figure 10: Sensitivity of Crash on α for our benchmarks.

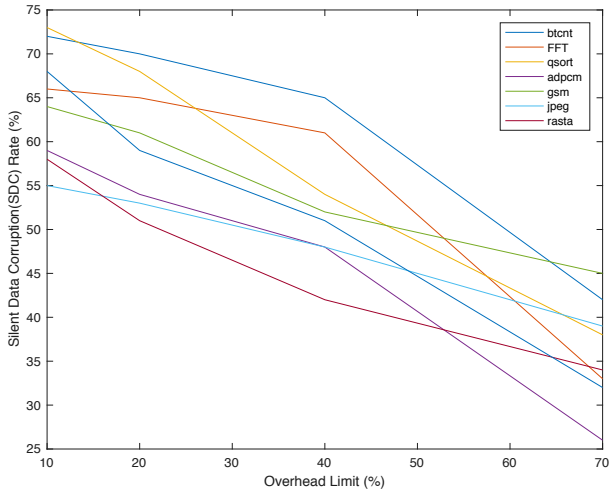

Figure 11: Sensitivity of Silent Data Corruption (SDC) on overhead limit for our benchmarks.

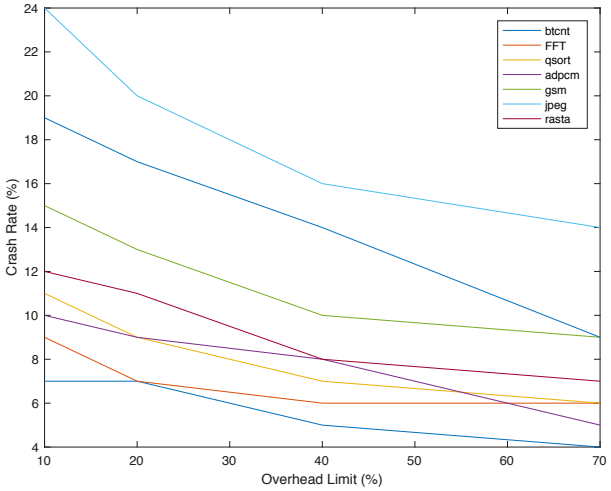

Figure 12: Sensitivity of Crash on overhead limit for our benchmarks.

α. This is because higher α values increase the significance of SDC in our instruction criticality formula. On the other hand, Figure 10 shows that Crash rates increase with increasing α. As expected, lower α increases the focus on Crash tolerance.

In the next set of experiments, we increase the overhead limit based on the dynamic execution of the application and keep a consistent rate of change each time. Specifically, we change the dynamic overhead limit with respect to total execution time of the BASE and conduct fault injection experiments on the benchmarks. Figures 11 and 12 show the SDC and Crash rates with increasing overhead limit ranging from 10%, to 70%. As can be seen from figures, both SDC and Crash rates tend to decrease with increasing overhead. A decrease of 10% in SDC and 5% in Crash rates is possible with only 20% dynamic overhead limit. On the other hand, when the overhead limit is increased to 70%, there was at least a 35% decrease in SDC rate and a 10% decrease in Crash rate. Crash rate tend to decrease slower since BASE has low crash rates in the beginning.

6 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 (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] A. Avizienis. 1985. The N-Version Approach to Fault-Tolerant Software. *IEEE Trans. Softw. Eng.* 11, 12 (Dec. 1985), 1491–1501. https://doi.org/10.1109/TSE.1985.231893

[2] N. R. Barraza. 2017. A Mixed Poisson Process and Empirical Bayes Estimation Based Software Reliability Growth Model and Simulation. In *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. 612–613. https://doi.org/10.1109/QRS-C.2017.114

[3] S. Chakravorty and L. V. Kale. 2007. A Fault Tolerance Protocol with Fast Fault Recovery. In *2007 IEEE International Parallel and Distributed Processing Symposium*. 1–10. https://doi.org/10.1109/IPDPS.2007.370310

[4] W. Excoffon, J. C. Fabre, and M. Lauer. 2017. Analysis of Adaptive Fault Tolerance for Resilient Computing. In *2017 13th European Dependable Computing Conference (EDCC)*. 50–57. https://doi.org/10.1109/EDCC.2017.22

[5] D. Fiala, F. Mueller, C. Engelmann, R. Riesen, K. Ferreira, and R. Brightwell. 2012. Detection and correction of silent data corruption for large-scale high-performance computing. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*. 1–12. https://doi.org/10.1109/SC.2012.49

[6] Brian A. Fields, Shai Rubin, and Rastislav Bodík. 2001. Focusing processor policies via critical-path prediction. In *ISCA*.

[7] Manish Gupta, David Roberts, Mitesh R. Meswani, Vilas Sridharan, Dean M. Tullsen, and Rajesh K. Gupta. 2016. Reliability and Performance Trade-off Study of Heterogeneous Memories. In *MEMSYS*.

[8] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. 2001. 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)*. 3–14. https://doi.org/10.1109/WWC.2001.990739

[9] A. L. Hopkins, T. B. Smith, and J. H. Lala. 1978. FTMP 8212;A highly reliable fault-tolerant multiprocess for aircraft. *Proc. IEEE* 66, 10 (Oct 1978), 1221–1239. https://doi.org/10.1109/PROC.1978.11113

[10] Chris Lattner and Vikram Adve. 2004. 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 (CGO '04)*. IEEE Computer Society, Washington, DC, USA, 75–. http://dl.acm.org/citation.cfm?id=977395.977673

[11] Chunho Lee, M. Potkonjak, and W. H. Mangione-Smith. 1997. MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of 30th Annual International Symposium on Microarchitecture*. 330–335. https://doi.org/10.1109/MICRO.1997.645830

[12] Tong Li, Alvin R. Lebeck, and Daniel J. Sorin. 2003. Quantifying instruction criticality for shared memory multiprocessors. In *SPAA*.

[13] Q. Lu, M. Farahani, J. Wei, A. Thomas, and K. Pattabiraman. 2015. LLFI: An Intermediate Code-Level Fault Injection Tool for Hardware Faults. In *2015 IEEE International Conference on Software Quality, Reliability and Security*. 11–16. https://doi.org/10.1109/QRS.2015.13

[14] R. E. Lyons and W. Vanderkulk. 1962. The Use of Triple-Modular Redundancy to Improve Computer Reliability. *IBM Journal of Research and Development* 6, 2 (April 1962), 200–209. https://doi.org/10.1147/rd.62.0200

[15] A. Mahmood and E. J. McCluskey. 1988. Concurrent error detection using watchdog processors-a survey. *IEEE Trans. Comput.* 37, 2 (Feb 1988), 160–174. https://doi.org/10.1109/12.2145

[16] N. Miskov-Zivanov and D. Marculescu. 2008. Modeling and Optimization for Soft-Error Reliability of Sequential Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27, 5 (May 2008), 803–816. https://doi.org/10.1109/TCAD.2008.917591

[17] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin. 2003. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.* 29–40. https://doi.org/10.1109/MICRO.2003.1253181

[18] P. C. Murley and G. R. Srinivasan. 1996. Soft-error Monte Carlo modeling program, SEMM. *IBM Journal of Research and Development* 40, 1 (Jan 1996), 109–118. https://doi.org/10.1147/rd.401.0109

[19] X. Ni and L. V. Kale. 2016. FlipBack: Automatic Targeted Protection against Silent Data Corruption. In *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*. 335–346. https://doi.org/10.1109/SC.2016.28

[20] N. Oh, P. P. Shirvani, and E. J. McCluskey. 2002. Error detection by duplicated instructions in super-scalar processors. *IEEE Transactions on Reliability* 51, 1 (Mar 2002), 63–75. https://doi.org/10.1109/24.994913

[21] Maurizio Rebaudengo, Matteo Sonza Reorda, Marco Torchiano, and Massimo Violante. 1999. Soft-Error Detection through Software Fault-Tolerance Techniques. In *DFT*.

[22] S. Rehman, F. Kriebel, M. Shafique, and J. Henkel. 2014. Reliability-Driven Software Transformations for Unreliable Hardware. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 33, 11 (Nov 2014), 1597–1610. https://doi.org/10.1109/TCAD.2014.2341894

[23] Semeen Rehman, Muhammad Shafique, and Jrg Henkel. 2016. *Reliable Software for Unreliable Hardware: A Cross Layer Perspective* (1st ed.). Springer Publishing Company, Incorporated.

[24] S. Rehman, A. Toma, F. Kriebel, M. Shafique, J. J. Chen, and J. Henkel. 2013. Reliable code generation and execution on unreliable hardware under joint functional and timing reliability considerations. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 273–282. https://doi.org/10.1109/RTAS.2013.6531099

[25] S. K. Reinhardt and S. S. Mukherjee. 2000. Transient fault detection via simultaneous multithreading. In *Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No.RS00201)*. 25–36.

[26] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. 2005. SWIFT: software implemented fault tolerance. In *International Symposium on Code Generation and Optimization*. 243–254. https://doi.org/10.1109/CGO.2005.34

[27] M. H. Schwarz, H. Sheng, B. Batchuluun, A. Sheleh, W. Chaaban, and J. BÃűrcsÃűk. 2009. Reliable software development methodology for safety related applications: From simulation to reliable source code. In *2009 XXII International Symposium on Information, Communication and Automation Technologies*. 1–7. https://doi.org/10.1109/ICAT.2009.5348447

[28] A. P. Singh and P. Tomar. 2013. A new model for Reliability Estimation of Component-Based Software. In *2013 3rd IEEE International Advance Computing Conference (IACC)*. 1431–1436. https://doi.org/10.1109/IAdCC.2013.6514437

[29] S. Subramaniam, A. Bracy, H. Wang, and G. H. Loh. 2009. Criticality-based optimizations for efficient load processing. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture*. 419–430. https://doi.org/10.1109/HPCA.2009.4798280

[30] R. Svenningsson, H. Eriksson, J. Vinter, and M. TÃűrngren. 2010. Model-Implemented Fault Injection for Hardware Fault Simulation. In *2010 Workshop on Model-Driven Engineering, Verification, and Validation*. 31–36. https://doi.org/10.1109/MoDeVVa.2010.11

[31] Eric Tune, Dean M. Tullsen, and Brad Calder. 2002. Quantifying Instruction Criticality. In *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques (PACT '02)*. IEEE Computer Society, Washington, DC, USA, 104–. http://dl.acm.org/citation.cfm?id=645989.674310

[32] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. 2014. 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)*. USENIX Association, Broomfield, CO, 249–265. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/yuan