Towards Analyzing and Improving Robustness of Software Applications to Intermittent and Permanent Faults in Hardware

Ankur Sharma‡, Joseph Sloan‡, Lucas F Wanner*, Salma H Elmalaki†, Mani B Srivastava‡, Puneet Gupta‡
UCLA Electrical Engineering Dept.‡, UCLA Computer Science Dept.*, UIUC Electrical and Computer Engineering Dept.†
Emails: {ankursharma, wanner, selmalaki, mbs, puneet}@ucla.edu‡∗, jsloan@illinois.edu†‡

Abstract—Although a significant fraction of emerging failure and wearout mechanisms result in intermittent or permanent faults in hardware, their impact (as distinct from transient faults) on software applications has not been well studied. In this paper, we develop a distinguishing application characteristic, referred to as similarity from fundamental circuit-level understanding of the failure mechanisms. We present a mathematical definition and a procedure for similarity computation for practical software applications and experimentally verify the relationship between similarity and fault rate. Leveraging dependence of application robustness on the similarity metric, we present example architecture independent code transformations to reduce similarity and thereby the worst-case fault rate with minimal performance degradation. Our experimental results with arithmetic unit faults show as much as 74% improvement in the worst case fault rate on benchmark kernels, with less than 10% runtime penalty.

I. INTRODUCTION

With the scaling of technology in the nanometer regime, increased process, voltage and temperature variations have exacerbated the infant mortality rate of VLSI ICs. Moreover, due to multiple aging and wearout induced hardware reliability loss mechanisms such as time dependent dielectric breakdown, hot electron injection, electromigration, negative bias temperature instability, and stress migration [2], it is expected that more components would suffer from unpredictable operational or in-field failures [26] [3] which initially manifest as intermittent faults and later develop into permanent faults [8] [7] [25]. Recent work [14] has shown that failures arising from process variations increasingly resemble the traditional permanent faults, i.e. the hardware’s erroneous behavior is a function of its state rather than time.

Existing software based fault tolerance mechanisms such as checkpointing and rollback [28], time redundant execution [19], recovery blocks [15], algorithm based fault tolerance [11] and executable assertions [20] exploit the impersistent nature of transient faults to either mask or detect and recover from them. Hence, they are inefficient to guard against the permanent and intermittent faults. Software monitors [12] for detecting in-field breakdowns rely on catastrophic software symptoms like application abort, kernel panic, etc. Although they are low cost solution, they do not provide coverage against silent data corruptions (SDC). Subsequent work [21] uses program invariants to detect SDC but they are susceptible to false positives. Authors in [9] [10] insert program-level detectors in SDC-hot sites to detect them. In this work, to reduce SDCs, we identify the sections of code that are more susceptible to large fault rates and propose architecture independent code transformations. The transformations proposed in this work complement other code optimizations [4] [24] [17] [18] employed to robustify codes. Our analysis is based on fundamental understanding of the circuit level fault models and hence, yields simple code transformations which incur minimal run-time overhead. We partly share our goal with [16] [30] where authors analyze the impact of intermittent faults on programs in terms of crashes and hangs, whereas we focus on SDC.

Our major contributions are

- We develop a code metric, called similarity, to quantify a code’s susceptibility to permanent and intermittent faults.
- We propose simple code transformations to reduce similarity and consequently, the worst case fault rate.

Although the theory and the conclusions presented in this work are applicable to any functional unit that accepts two operands as inputs, we inject and study faults only in the multiplier as they cause difficult to detect SDCs more often than faults in other units like the adder. Authors in [1] study the detection of permanent faults in multipliers for the same reason.

The paper is organized as follows: Section II presents the fault models followed by a discussion on the similarity metric and the code transformations in the Section III. In Section IV, experimental setup and results are described and we conclude in Section V.

II. FAULT MODELS

Firstly, we define some commonly used terms. An input vector, $v$, is a bit vector feeding the inputs of a hardware unit. Faulty run refers to the tuple $\{A, I, F\}$ - an application $A$ executing an input $I$ on a hardware with fault $F$. Fault rate is the fraction of input vectors that activate the fault amongst all the input vectors that access the faulty hardware in a faulty run.

In this section, we’ll present the permanent and the intermittent fault models used in this study and discuss how they are distinct from the transient fault model.

Various failure mechanisms, depending upon their circuit level impact, have been modeled at the gate-level as stuck-at(0,1) or delay faults [7]. While permanent faults, remain active throughout the execution of the application, intermittent faults have been assumed to activate in the beginning and are characterized by an activation period ($t_a$), an idle period ($t_i$), and a burst length ($t_{burst}$) - number of times the activation-idle cycle repeats, as shown in Fig.1.

Transient fault activation which is usually modeled as a single event upset is time dependent. At the gate-level, a fault is said to be activated if any one of the input or the output bits of the faulty gate is flipped. Whereas, since permanent faults perpetually exist, their activatation is solely dependent on the system state, independent of
Assume that in an execution, $N$ input vectors, each composed of two operands, access a faulty unit. Then, similarity is defined as,

$$S \sim \frac{1}{N^2} \sum_{i=1}^{N} \sum_{j=1, j \neq i}^{N} s_{ij}$$  \hspace{1cm} (1)$$

where, $s_{ij}$ is the number of non-constant operands shared between the $i^{th}$ and the $j^{th}$ input vectors. Non-constant operands refer to those operands whose values are independent of the input to the application. Mathematical justification for constant operand sharing not contributing to the similarity is explained in [23].

Using the procedure outlined in Eqn.(1), similarity which is inherent to a code, can be statically computed. For instance, consider the kernel shown in Fig.2. $N$ input vectors share the first operand with the value $e$. Thus, there are $N(N-1)$ pairs sharing an operand.

Therefore, for this original code (Org), $S_{Org} = \frac{N-1}{N^2}$. Drawback of static computation is that if there is no compiler visible operand sharing in the source code then statically computed similarity will be zero. To account for the contribution of application inputs, similarity averaged over several fault-free profiled executions needs to be computed. Since one of the benchmark kernels has zero compiler visible operand sharing amongst multiply instructions, profiling has been used to compute similarity.

To reduce operand sharing, we propose two architecture independent code transformations, referred to as Swap (Sw) and Swap-Negate (SwN). For the original code of Fig.2, both the transformations are shown in Fig.4a-b, respectively. In Sw, for-loop is divided into two equal halves, each half increments by 2 and in one of the loops operands are swapped. As a result, there are only half as many input

because due to small overscaling factors, only very specific pairs of input vectors sensitize critical paths.

A clear implication of this observation is that if a code execution generates a lot of input vectors that share at least one operand then the fault rate would tend to be either very large or very small. To summarize, operand sharing implies large amount of bit sharing which implies high CFP which implies large standard deviation in the fault rate, $\sigma_{FR}$ (computed over several faulty runs for a given A and F) which implies large worst case fault rates, $\omega_{FR} (= \mu_{FR} + 3\sigma_{FR})$, assuming $\mu_{FR}$ remains same. $\mu_{FR}$ is the mean fault rate. In [23] we have mathematically derived $\sigma_{FR}$ as a function of CFP.

### III. Similarity Metric and Code Transformations

In this section, we define similarity, to quantify the amount of operand sharing in a given piece of high-level code and then propose code transformations to reduce it. According to our claim, that should reduce $\sigma_{FR}$. We are targeting fault rate because the energy benefits derived from timing speculative architectures [22] critically require low fault rate. Similarly, the performance penalty due to recovery and reprocessing in checkpointing based fault tolerance mechanisms is directly proportional to the fault rate [27].

Assume that in an execution, $N$ input vectors, each composed of two operands, access a faulty unit. Then, similarity is defined as,

$$S \sim \frac{1}{N^2} \sum_{i=1}^{N} \sum_{j=1, j \neq i}^{N} s_{ij}$$  \hspace{1cm} (1)$$

where, $s_{ij}$ is the number of non-constant operands shared between the $i^{th}$ and the $j^{th}$ input vectors. Non-constant operands refer to those operands whose values are independent of the input to the application. Mathematical justification for constant operand sharing not contributing to the similarity is explained in [23].

Using the procedure outlined in Eqn.(1), similarity which is inherent to a code, can be statically computed. For instance, consider the kernel shown in Fig.2. $N$ input vectors share the first operand with the value $e$. Thus, there are $N(N-1)$ pairs sharing an operand.

Therefore, for this original code (Org), $S_{Org} = \frac{N-1}{N^2}$. Drawback of static computation is that if there is no compiler visible operand sharing in the source code then statically computed similarity will be zero. To account for the contribution of application inputs, similarity averaged over several fault-free profiled executions needs to be computed. Since one of the benchmark kernels has zero compiler visible operand sharing amongst multiply instructions, profiling has been used to compute similarity.

To reduce operand sharing, we propose two architecture independent code transformations, referred to as Swap (Sw) and Swap-Negate (SwN). For the original code of Fig.2, both the transformations are shown in Fig.4a-b, respectively. In Sw, for-loop is divided into two equal halves, each half increments by 2 and in one of the loops operands are swapped. As a result, there are only half as many input
TABLE I: Similarity values (in the order of 1e-3) for the original (Org), swapped (Sw) and swap-negated (SwN) codes of all the applications are shown. Normalized (with respect to Org) similarity values are shown in the brackets. Number of faulty input vectors, \( N \sim 42000 \)

<table>
<thead>
<tr>
<th>App</th>
<th>Fr</th>
<th>Ac</th>
<th>Mm</th>
<th>Km</th>
<th>Ft</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S_{Org} )</td>
<td>14.9</td>
<td>5.50</td>
<td>4.70</td>
<td>0.40</td>
<td>0.04</td>
</tr>
<tr>
<td>( S_{Sw} )</td>
<td>7.95(0.53)</td>
<td>4.36(0.79)</td>
<td>4.31(0.91)</td>
<td>0.38(0.96)</td>
<td>0.04(1.00)</td>
</tr>
<tr>
<td>( S_{SwN} )</td>
<td>7.94(0.53)</td>
<td>3.28(0.59)</td>
<td>4.30(0.91)</td>
<td>0.37(0.96)</td>
<td>0.04(1.00)</td>
</tr>
</tbody>
</table>

which is approximately, 2X smaller than \( S_{Org} \).

SwN is an improvement over Sw to handle those faults that are immune to operand order. SwN not only swaps the operands but also multiplies them by -1. In presence of loop-carried dependences, the two for-loops should be interleaved to maintain the iteration order.

Table I compares the similarity of the transformed codes with the original codes for five different applications - FIR filter (Fr), Autocorrelation (Ac), Matrix Multiplication (Mm), Kmeans Clustering (Km) and FFT (Ft). There is orders of magnitude difference between the absolute similarity of Fr (14.9e-3) and Ft (0.04e-3). Since the original code of Fr has relatively very large compiler visible operand sharing, transformations cause significant reduction. Whereas, in case of Km and Ft, operand sharing is minimal, mainly due to the value of the inputs. Hence, transformations could not systematically reduce it further. Moreover, compiler does not preserve all the transformations which are applied at the high-level.

IV. Experimental Setup and Results

We inject faults in architecture visible multiply instructions using VarEmu [29], an instruction-level emulator. A gate-level timing simulator, Mentor ModelSim, back-annotated with the standard delay format (SDF) file from the logic synthesis is coupled with VarEmu. It is selectively and on-demand invoked to accurately model the architecture-level manifestations of gate-level delay faults, as also done in [13]. For stuck-at faults, however, to achieve the same accuracy while not incurring the run-time overhead due to communication with an external simulator, we translated every single gate-level synthesized netlist with a unique fault injected into it, into a C++ equivalent and linked it with the VarEmu.

We used Cadence RTL Compiler synthesized 32-bit multiplier design for fault injection. While for delay fault injection we overscaled the frequency by 20%, for stuck-at fault, 300 nets were randomly chosen and a fault was injected in one of these locations. Interrupted fault model parameters are as follows: \( t_d = t_l = 100 \) instructions, \( \text{burst} \sim \{50, 500, 2500\} \) cycles. We study the efficacy of code transformations under three different burst lengths. We experimented with three fault models: Permanent/Stuck-at (PS), Permanent/Delay (PD) and Interrupted/Stuck-at (IS). Since focus of this work is to study the impact of hardware faults on SDC, faults in OS which often has some detectable catastrophic impact were not injected.

**PS fault model:** There are two important observations:

- Both the transformations reduce \( \sigma_{FR} \) which follows the reductions in the similarity (see Table II). While reduction is maximum for Fr (\( S \) reduces to 0.53X and \( \sigma_{FR} \) reduces to 0.57X by SwN), it is negligible for Km and Ft.
- Subject to the reduction in \( \mu_{FR} \), reduction in \( \sigma_{FR} \) implies reduction in \( \omega_{FR} \). For Mm, maximum improvement of 74% in \( \omega_{FR} \) is observed due to combined reduction in \( \mu_{FR} \) as well as \( \sigma_{FR} \).

**IS fault model:** Table III reports reduction in \( \sigma_{FR} \) and \( \omega_{FR} \) due to SwN transformation. Results due to Sw transformation (not shown in the table) are very similar to SwN. There are two main observations:

- For less enduring faults (IS0), even significant reduction in similarity may not effect any reduction in \( \sigma_{FR} \) because as the fault duration reduces, fault activation is more time dependent. Consequently, for an instance, if two instructions which are separated by many instructions generate identical input vectors and one of them activates the fault, then other would not activate if the separation between them is more than the fault duration. Therefore, transformations can’t guarantee systematic reduction in \( \sigma_{FR} \), as evident from the column under IS0 in Table III.
- As fault duration rises (IS1 \( \rightarrow \) IS2), results consistently improve.

**PD fault model:** Table IV shows results for 3 out-of-5 applications because in Km and Ft, no timing violations were observed. In Ac, Fr and Mm, although transformations reduce \( \sigma_{FR} \) as well as \( \omega_{FR} \), reductions vary with designs. We experimented with DC synthesized multiplier design using “carry-save-array” synthesis model and much larger reductions were observed. Last two columns in the same table show that for Fr, \( \omega_{FR} \) reduced by 55X. This was effected due to 64X reduction in \( \sigma_{FR} \) and 17X reduction in \( \mu_{FR} \). We observed remarkable design dependence for delay faults because timing violations depend on critical path sensitization which depends on very specific input vector sequence which vary with designs.

Although the results show that the transformations improve \( \mu_{FR} \), we need to understand the dynamics between \( \mu_{FR} \) and the similarity.
It demands more careful analysis because $\mu_{FR}$ is a strong function of inputs and the fault location rather than being just inherent to an implementation. Though we applied our code transformations by modifying the high-level code, compiler can likely do a better job by applying transformations automatically since it can preserve them through the compiler optimizations.

Both the transformations have minimal performance overhead of $<10\%$ as recorded in the Table V. For estimating performance overhead, benchmarks were executed on the host machine (x86_64) instead of VarEmu because due to emulation overhead, runtime numbers obtained from VarEmu are not meaningful.

V. CONCLUSIONS AND FUTURE WORK

In this work, we have studied the impact of permanent and intermittent hardware failures on programs in terms of fault rate and based on our analysis developed a code metric, similarity, that correlates with the standard deviation in the fault rate. Leveraging this dependence, we have proposed architecture independent code transformations to reduce similarity and thus, curb the worst-case fault rates by as much as 74%. We conclude that similarity as a code metric can be reliably applied to model standard deviation in the fault rate for permanent stuck-at fault model and intermittent faults with long duration. In case of delay faults due to heavy design dependence the correlation is comparatively weaker. More details on this work can be found in [23]. In the future, we would like to 1) study the impact of similarity on average fault rate, 2) explore architecture dependent code transformations, and 3) apply the proposed transformations at the compiler level.

VI. ACKNOWLEDGEMENT

Authors would like to thank Prof. Rakesh Kumar from UIUC for his valuable suggestions and comments.

REFERENCES