University of California
Los Angeles

Understanding Software Application Behaviour in Presence of Permanent and Intermittent Hardware Faults

A thesis submitted in partial satisfaction
of the requirements for the degree
Master of Science in Electrical Engineering

by

Ankur Sharma

2013

© Copyright by

Ankur Sharma

2013

Abstract of the Thesis

Understanding Software Application Behaviour in Presence of Permanent and Intermittent Hardware Faults

by

Ankur Sharma

Master of Science in Electrical Engineering

University of California, Los Angeles, 2013

Professor Puneet Gupta, Chair

Over past three decades technological advancement in fabrication of VLSI ICs has been accompanied by shrinking of device sizes and scaling of supply voltage. While power, area and performance have constantly improved, hardware reliability is becoming a growing concern. Due to increased process, voltage and temperature (PVT) variations, the infant mortality rate has gone up. Coupled with PVT variations, aging and wearout induced failures have exacerbated the problem as devices unexpectedly fail while in operation. Although a significant fraction of emerging failure and wearout mechanisms result in intermittent or permanent faults in the hardware, their impact (as distinct from transient faults) on software applications has not been well studied. In this work, we analyze the impact of such failures on software applications and develop a distinguishing application characteristic, referred to as *similarity* from basic circuit-level understanding of the failure mechanisms. We present a mathematical definition and approximations for similarity computation for practical software applications and experimentally verify the relationship between similarity and fault rate. Leveraging the dependence of application robustness on similarity metric, we present example architecture independent code transformations to reduce similarity and

thereby the worst case fault rate with minimal performance degradation. The experiments with arithmetic unit faults show as much as 74% improvement in the worst case fault rate on benchmark kernels with less than 10% performance degradation.

The thesis of Ankur Sharma is approved.

Lara Dolecek

Mani B Srivastava

Puneet Gupta, Committee Chair

University of California, Los Angeles

2013

To my parents

Table of Contents

1 Introduction **1**

1.1 A Review 2

1.2 Thesis Outline 7

2 Modeling Fault and Fault Rate **9**

2.1 Fault Models 9

2.2 Analytical Modeling of Fault Rate 16

2.3 Pictorial Representation of Fault Rate 21

2.4 Chapter Summary 21

3 Similarity and Code Transformations **23**

3.1 Practical Approximation to Similarity 23

3.2 Code Transformation 28

3.3 Chapter Summary 32

4 Experimental Setup and Results **33**

4.1 Experimental Setup 33

4.2 Results 37

4.3 Chapter Summary 46

5 Conclusions **48**

5.1 Conclusions 48

5.2 Future Work 49

References **52**

List of Figures

2.1 Intermittent fault model parameters, adapted from [GSB08] 11

2.2 A general 2-bit multiplier logic block 12

2.3 Average conditional failure probability (CFP, $\gamma_{avg}(k)$) as a function of the number of bits shared (k) between input vectors. Stuckat faults are injected in three different locations (L) of three different designs (D) of 8-bit multipliers. 14

2.4 Average conditional failure probability (CFP, $\gamma_{avg}(k)$) as a function of the number of bits shared ($k > 15$) between two pairs of input vectors. For lower values of k, the curve almost remains flat. 16

2.5 (a) Subset of fault activating input vectors (b) Input vectors generated by a faulty run of Fr application (c) overlap between the (a) and (b) (shown in red). 21

3.1 CFP as a function of the number of operands shared (*op*), as obtained from verilog simulations on 8-bit multiplier designs (see Section 2.1). . 26

3.2 An operand being shared between two input vectors, namely v_1 and v_4 implies that there exist two pairs of two consecutive input vectors, namely, $\{\{v_0, v_1\}\{v_3, v_4\}\}$ and $\{\{v_1, v_2\}\{v_4, v_5\}\}$, each sharing an operand. 26

3.3 A general 2-bit multiplier logic block 28

3.4 An example code in its original version. 28

3.5 Code of Fig.3.4 after applying transformation *Swap* (Sw). Half the operands are swapped. 29

3.6 Code of Fig.3.4 after applying transformation *Swap-Negate* (SwN). Half the operands are swapped and multiplied by -1. 30

3.7 Input vectors generated in a faulty run of Fr's original (a) and transformed codes - Sw (b) and SwN (c). 32

4.1 (a) VarEmu architecture for fault model emulation. (b) Enabling faults using VarEmu based fault model implementation 36

4.2 Comparing the efficacy of code transformations Sw and SwN, under optimization flags O3 and O0. (a) Reduction in standard deviation in fault rate $\sigma_{\mathcal{FR}}^{norm,avg}$, (b) Reduction in strict similarity S_s. 44

5.1 Overlap (red region) between the fault activating input vectors (blue region) and the input vectors generated (green region) by the original (left column), Sw transformed (middle column) and SwN (right column) transformed codes for three different faults - $F1$, $F2$ and $F3$. 50

List of Tables

2.1 Frequently used notations 10

2.2 Frequently used acronyms 10

2.3 Frequently used phrases 11

2.4 Conditional Failure Probability (CFP) as a function of the number of bits guranteed to be shared (k), for fault $z0$ stuck-at 0 in the 2-bit multiplier logic. As k increases, CFP increases exponentially. $k = 0$ implies atleast 0 bits are shared which means they can be any two 4-bit input vectors. 13

2.5 Benchmark kernels: Count of multiply instructions, output type and correlation between fault rate and error magnitude as obtained from fault injection experiments with permanent stuck-at fault model. 16

3.1 Notations introduced in this chapter 23

3.2 Acronyms introduced in this chapter 24

3.3 Relaxed (S_r) and strict (S_s) similarity values for the original (Org), swapped (Sw) and swap-negated (SwN) codes of all the applications. Similarity values normalized with respect to Org are shown in the brackets. 30

4.1 Frequently used notations 34

4.2 Frequently used acronyms 34

4.3 Permanent/Stuck-at fault model: Correlating reduction in similarity (S_s) and reduction in $\sigma_{\mathcal{FR}}$ and $\sigma_{\mathcal{FR}}/\mu_{\mathcal{FR}}$ due to Sw and SwN transformations. Similarity values are normalized with respect to the original code. 39

4.4 Intermittent/Stuck-at fault model: Table shows average normalized values of $\sigma_{\mathcal{FR}}$ and $\sigma_{\mathcal{FR}}/\mu_{\mathcal{FR}}$ due to SwN transformation for three different burst lengths ($l_{burst} = 50, 500, 2500$), corresponding to IS0, IS1 and IS2, respectively. Values are normalized with respect to the original code values corresponding to the respective burst length. 40

4.5 Permanent/Delay fault model: Results from experiments on two different designs - one synthesized by the RC and the other synthesized by the DC. Table shows average normalized values of $\sigma_{\mathcal{FR}}$ and $\sigma_{\mathcal{FR}}/\mu_{\mathcal{FR}}$ due to Sw and SwN transformation. NTV refers to “No Timing Violations”. 40

4.6 Permanent/Stuck-at fault model: Average reduction in $\omega_{\mathcal{FR}}$ and $\omega_{\mathcal{EM}}$ due to Sw and SwN transformations compared to the original code. Maximum reduction of 74% is observed for Mm (shown in bold font). . 42

4.7 Intermittent/Stuck-at fault model: Average reduction in $\omega_{\mathcal{FR}}$ and $\omega_{\mathcal{EM}}$ due to SwN transformation compared to the original code. IS0, IS1, IS2 correspond to three different burst lengths ($l_{burst} = 50, 500, 2500$). . . . 43

4.8 Permanent/Delay fault model: Results from experiments on two different designs - one synthesized by the RC and the other synthesized by the DC. Average reduction in $\omega_{\mathcal{FR}}$ and $\omega_{\mathcal{EM}}$ due to Sw and SwN transformations compared to the original code. NTV refers to “No Timing Violations”. 43

4.9 Fault injection in adder instructions using permanent stuck-at fault model. Values inside the brackets are normalized (with respect to the original code, Org) strict similarity (S_s). 46

4.10 Runtime penalty in transformed codes. $R_{Sw,norm}$ and $R_{SwN,norm}$ are the runtimes of the transformed codes normalized with respect to the original code's runtime. 46

Acknowledgments

This thesis would not have been possible without the constant guidance and support of my advisor, Prof. Puneet Gupta. Our frequent technical discussions have been very fascinating and educative. I would like to thank Prof. Mani Srivastava at UCLA, Prof. Rakesh Kumar at UIUC and Joseph Sloan at UIUC for their valuable suggestions and constant inputs.

I would like to acknowledge the support of NESL lab members. Discussions and technical support provided by Lucas Wanner and Salma Elmalaki was very crucial in the successful completion of this project. My colleagues at NanoCAD lab over the last two years: Abde Ali, John Lee, Rani Ghaida, Liangzhen Lai, Tanaya Sahu, Shaodi Wang, Mukul Gupta, Mark Gottscho and Yasmine Badr have all been very helpful. Time spent with them all would always be a memorable experience. I would also like to express my thanks to my parents and my brother for their unconditional support and encouragement.

Lastly, I would like to express my gratitude to the Almighty Supreme Lord for bestowing upon me His choicest blessings and giving me the ability to think, to analyze and to express.

CHAPTER 1

Introduction

With the scaling of technology in the nanometer regime, increased process, voltage and temperature (PVT) variations have exacerbated the infant mortality rate. Due to insufficent burn-in defective devices inevitably reach the end user [Bor05]. Moreover, due to multiple aging and wearout induced hardware reliability loss mechanisms, it is expected that more components would suffer from unpredictable operational or in-field failures [SAB04] [BSO05]. Wearout induced failures initially manifest as intermittent faults and later develop into permanent faults [GSB08] [GAM02] [SGH07]. Manufacturing residues may also result in intermittent contacts [GSB08]. Recent work in [NKS12] 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. This persistent nature of permanent and intermittent hardware faults as opposed to the transient hardware faults, renders existing software based reliability mechanisms either inefficient or inapplicable.

In this work, we study the impact of intermittent and permanent hardware faults on software programs with a goal to robustify the latter against the former, while incurring minimal run-time overhead. We observe that certain code sections are more susceptible to such faults i.e., they can activate much larger number of faults compared to the other sections of the code. Based on the basic circuit-level understanding of failure mechansims we develop a code metric called "similarity" to quantify its susceptibility to such faults. Leveraging their relationship, we

propose simple code transformations to reduce the similarity and thereby, optimize the worst case fault rate.

1.1 A Review

In this section, we review the commonly known device level failure mechanisms. Then we relate a representative subset of the existing work on software based fault tolerance mechanisms including symptom based detection of hardware faults, detection and reduction of silent data corruptions (SDC) and code optimizations to improve the resiliency of the software program. Since SDC's are the toughest to detect and recover from, our work focusses on reducing SDC's through simple code transformations.

1.1.1 Hardware Failure Mechanisms

Time dependent dielectric breakdown (TDDB), also known as oxide breakdown refers to the gradual formation of the conductive path (wearing out) through the oxide layer. With decreasing oxide thickness, $< 2nm$ now a days, the gate leakage current is exponentially growing [Sta02]. Apart from increasing the power consumption, gate current stresses the oxide eventually leading to breakdown. While stress induces long-term parameter shifts which affect the power and the performance of the device, oxide breakdown can even render it nonfunctional depending upon the post-breakdown conduction.

Electromigration (EM) describes the atomic movement of metallic material under electric field. Its a diffusive phenomena that creates voids and protrusions causing opens (high resistance) and shorts in the interconnect. With decreasing line-widths and increasing current densities, the mean time to failure is increasing [HK89] [HRR99]. Stress migration (SM) is very similar to EM in terms of physical phenomena and its circuit-level impact. While EM is caused by electric field, SM

is caused due to mechanical stress gradient [Ass03].

Hot carrier injection (HCI) refers to the injection of the "hot" carriers from the substrate or the channel into the thin oxide gate, thereby heating up the device and increasing the gate leakage at the cost of ON-current. Hot carriers can also break Si-H bonds at the interface leaving behind dangling Si bonds that form the interface traps. Oxide charges and interface defects produce threshold voltage shift, transconductance degradation, drain current reduction, etc., and eventually lead to device failure [MPR00].

With buried channel devices while HCI was the dominant concern, due to increased field, temperature and use of surface channel devices, negative bias temperature instability (NBTI) begins to become an issue [HDP06] [CSG11]. NBTI is observed in pMOS transistors under negative bias. It is attributed to two types of traps - interface traps due to dangling Si bonds, similar to HCI, and pre-existing oxide traps. While both the traps degrade threshold voltage, degradation due to oxide traps can be recovered upon removing the negative bias.

1.1.2 Software Based Fault Tolerance Mechanisms: Masking, detection and recovering from soft errors

In order to tolerate hardware failures various hardware and software based solutions have been proposed in the literature. While hardware based solutions are more effective in terms of coverage, detection latency etc., they usually come at a high penalty - either in area/power or performance. For flagship processors requiring high availablity, such solutions might be needed, but for commodity processors cheaper software based solutions are preferred where 100% coverage is not a requirement. Most of the existing software approaches are targetted at transient hardware faults and make use of their impersistent nature.

Saha [Sah] presents a concise compilation of software techniques for fault tol-

erance. Some of the most commonly used techniques include checkpoint and roll back, time redundant execution, recovery blocks, algorithm based fault tolerance, executable assertions. These techniques either mask the failures or detect and recover from them. Checkpointing based techniques [WHV95] periodically save the system state, upon detection of an error roll back to the latest correct checkpoint and re-execute. Checkpointing can be very effective when faults transiently appear. With persistent faults large number of roll backs and re-executions will be required. If the fault is permanent, then re-executions do not help at all. Redundancy based techniques like N-Version Programming [Avi85] and recovery blocks [Ran75] leverage design diversity in the software to provide fault tolerance, albeit with high run-time penalties. Time redundant execution techniques perform redundant computations either at instruction level, procedure level, or program level to detect errors by comparing the results of duplicate executions [HLD05] [OSM02] [OM01] [RCV05] [CRA06]. These schemes assume impersistent nature of the failures and hope for different corruption signatures in the redundant executions to enable detection. However, intermittent and permanent failures can possibly lead to the same corruption signatures in the redundant executions because faults persist in the same location for longer periods. Permanent fault detection through time redundant task executions has been proposed in [AFK03]. Authors inject permanent faults at some point in the application; if that fault appears during the task execution and affects the redundant tasks in 'detectably distinct' ways then it can be detected. The probability of detection reduces as the task size reduces. Algorithm based fault tolerance is achieved by encoding the input data and modifying the algorithm to operate on the encoded data [HA84] [CD06] [CFG05]. For very specific applications like matrix multiplication, it provides for self-checking. Assertion based detection schemes apply checks on the control flow or derive program specific invariants to trap deviation from the normal behavior [Sah06] [GRS03] [And79].

1.1.3 Software monitors for in-field breakdown and SDC detection

Symptom based detection mechanisms have emerged as very effective low cost solutions. Paul *et. al.* [WP06] propose several fault screeners to detect perturbations from the expected result of an instruction and implement these screeners at architecture level in the hardware. Fault screeners, essentially, maintain a history of past outputs by an instruction and compare the current result with the previous outputs to detect any deviation. Authors use a random bit flip transient fault model in their injection experiments. ReStore [WP06] is a combination of checkpointing and symptom detection to safeguard the system against soft errors. It detects symptoms like memory exceptions, incorrect control flow etc., and rolls back to the last correct checkpoint. As discussed above, in presence of intermittent and permanent faults, large number of roll backs and re-execution can totally eclipse the benefits of this technique. SWAT [LRS08a] [LRS08b] is a software monitor for detecting in-field breakdowns. It can detect fatal software symptoms like application abort and kernel panic but provides no coverage against SDC's. In a subsequent work [SLR08] authors make use of range based program invariants to detect SDC's in presence of permanent faults. While SDC's are reduced by 74%, such program invariants are highly application specific and suffer from high false positives [HAN12a], thereby severly affecting the performance. Other hardware / software mechansims for online diagnosis of hardware defects have been presented in [BSO05] [CMA07].

Since SDC's do not leave behind any failure trace, they represent the worst-case scenario. In addition, SDC rate in unbounded. Nonetheless, SDC detection capabilities are required for broad adoption of symptom based detection mechanisms. Therefore, recently published works [HAN12b] [HAN12a] [FGA10] focus on detecting and reducing SDC's, specifically due to transient faults. Shoestring [FGA10] assumes all writes to the memory and function arguments are SDC causing sites and uses instruction duplication to detect them. In [HAN12a] and

[HAN12b] authors generate more thorough application reliability profile using Relyzer [HAN12b], identify SDC-hot sites at instruction level granularity and then insert program level detectors in those specific locations only.

1.1.4 Program reliability metric and code optimizations

With the aim of improving the reliability of the software against faults, in the past, researchers have studied the impact of various fault models on software programs and/or proposed code optimizations either at compiler level or at algorithm level. [SK09] proposes program vulnerability factor to capture the resiliency of a program to soft errors as independent from the hardware. Pattabiraman *et. al.* [PKI05] developed metrics to identify critical application variables that need to be protected with the aim to reduce system crashes and limit fault propagation. Same authors study the impact of intermittent faults on applications [RPG10] [WRP11] and conclude that because intermittent faults are very less likely to be masked compared to the transient faults existing solutions to tolerate latter may not be applicable to the former. Rehman *et. al.* [RSK12] [RSK11] perform static estimations of the software's reliability and trade-off performance for reliability through compiler directed code optimizations. They study the occupancy of instructions in the pipeline stages and the life of variables in a program. Using the arguments that instructions occupying pipeline stages for longer durations and long-living variables are more vulenrable to soft errors they propose instruction rescheduling techniques to improve the software's reliability. Cho *et. al.* [CLM12] add application specific detectors to recover from soft errors. Sloan *et. al.* propose algorithmic code optimizations by utilizing the fact that iterative applications like optimization problems, are more error-resilient than non-iterative applications. Hence, they reformulate example problems as stochastic optimization problem and employ the stochastic solver to solve them with the penalty of as much as 1000X increase in the instruction count. They validate their approach

against random bit flip fault model which does not capture the persistent nature of intermittent and permanent faults.

1.2 Thesis Outline

In this work, we study what sections of code can cause more SDC's in presence of permanent and intermittent faults and how can the SDC rate be reduced. Our proposed architecture independent code transformations come at $< 10\%$ runtime penality in the worst-case and complement the above techniques.

Our major contributions are

- We derive a mathematical model for fault rate to explain why permanent and intermittent faults impact different applications differently.
- We develop a code metric called *similarity* to quantify its susceptibility to such 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 with two operands as inputs, we inject and study faults in multiplier as they cause difficult to detect SDCs more often. Authors in [AFK03] study detection of permanent faults in multipliers for the same reason. Although we briefly mention the results with adder units, most of the executions result in crashes or infinite execution that can be detected and diagnosed using low-cost software monitors [LRS08a].

This thesis is organized as follows. Chapter 2 presents the fault models and the analytical modeling of fault rate. This chapter contributes valuable insights into the distinctive nature of permanent and intermittent fault models that are lever-

aged later to formally define similarity. Chapter 3 develops a practical approximation to efficiently compute similarity and presents code transformations. Chapter 4 describes our hybrid fault simulation/injection infrastructure and discusses the impact of code transformations on fault rate. Finally, Chapter 5 concludes the thesis.

CHAPTER 2

Modeling Fault and Fault Rate

In this chapter we'll discuss the gate-level fault models for permanent and intermittent faults used in this work and then, based on the charatersitics of these models, analyze how persistent nature of these faults distinguishes them from the transient faults. Then, we'll analytically model the fault rate.

2.1 Fault Models

Various failure mechanisms, depending upon their circuit level impact, are classified into two categories - mechanisms that cause shorts or opens in gates or interconnects, like EM, and mechanisms that slow down a gate, like NBTI [GAM02] [GGL02]. Former can be modeled at the gate-level as a *stuck-at(0,1)* fault or a *bridging* fault. Since stuck-at fault model is the most widely used fault model, we model the failures due to EM and SM by a stuck-at(0,1) fault. Slowing down of a gate manifest itself as timing violation when the output of that gate transitions and it lies on one of the critical paths. Hence we model the failures due to mechanisms like NBTI, TDDB and HCI, as delay faults.

While permanent faults, once activated, persist for the entire lifetime of the device, intermittent faults - either stuck-at or delay, are characterized by an activation period (t_a) - when fault is active, an idle period (t_i) - when fault is inactive, and burst length (l_{burst}) - number of times the activation-idle cycle repeats, as shown in Fig.2.1. t_a and t_i as mentioned in [GSB08] are in terms of CPU cy-

$\{A, I, F\}$	Application, Input, Fault tuplet
t_a	Duration of active cycle for intermittent faults
t_i	Duration of idle cycle for intermittent faults
l_{burst}	Burst length - number of active-idle cycles
v	Input vector
$\{a, b\}$	Two operands a and b of the input vector v
$h_F(v_i)$	Fault function for fault F. Its a boolean expression such that if v satisfies it, then it activates the fault.
$\mathcal{X}_i$	Bernouli random variable to indicate i^{th} input vector, v_i, activates a fault
p_i	Failure probability associated with v_i
N	Total number of instructions accessing the faulty hardware
$\mathcal{FR}$	Fault rate random variable
$\mathcal{EM}$	Error magnitude random variable
$\mu_{\mathcal{FR}}$	Fault rate for a given F and A, averaged over all I
$\sigma_{\mathcal{FR}}$	Standard deviation in fault rate for a given F and A, computed over all I
$\omega_{\mathcal{FR}}$	Worst case fault rate defined as $\mu_{\mathcal{FR}} + 3 * \sigma_{\mathcal{FR}}$
$\omega_{\mathcal{EM}}$	Worst case error magnitude
γ_{ij}	Conditional failure probability for the pair of input vectors v_i and v_j

Table 2.1: Frequently used notations

SDC	Silent Data Corruption
CFP	Conditional Failure Probability
Fr, Ac, Mm, Km, Ft	Benchmark kernels: Fir, Autocorrelation, Matrix Multiplication, Kmeans clustering, FFT

Table 2.2: Frequently used acronyms

Input vector v	It is a bit vector feeding the inputs of a hardware unit.
Fault activating input vector	An input vector that activates a given fault.
Faulty run	Refers to the tuplet $\{A, I, F\}$ - an application A executing an input I on a hardware with fault F.
Fault rate	The fraction of input vectors that activate the fault amongst all the input vectors that access the faulty hardware in a faulty run.

Table 2.3: Frequently used phrases

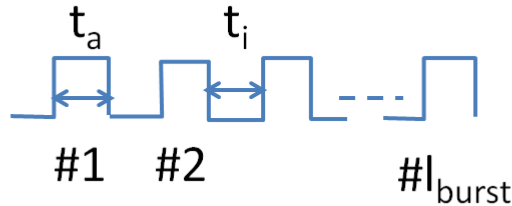

Figure 2.1: Intermittent fault model parameters, adapted from [GSB08]

cles, in this work, however, we have assumed CPI (cycles per instruction) of 1 and treat the parameters in terms of instructions. Since we'll be doing functional simulations this assumption does not affect our results or conclusions.

2.1.1 Distinctive nature of permanent/intermittent stuck-at fault model

Stuck-at faults are activated by certain input vectors only. For instance, consider the 2-bit multiplier logic block shown in Fig.2.2. If primary output $z0$ is stuck-at 0, it can be activated only when $a0 = b0 = 1$. In other words, a stuck-at fault, F, in some arithmetic unit that processes two operands a and b can be characterized by a boolean expression referred to as *fault function*, $h_F(a, b)$ such that

$$h_F(a, b) = 1 \Leftrightarrow \{a, b\} \; activate \; the \; fault \; F$$

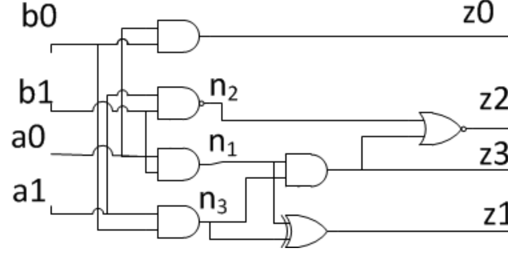

Figure 2.2: A general 2-bit multiplier logic block

Therefore, for the fault $z0$ stuck-at 0, in the example above,

$$h_{z0=0}(a, b) = a0b0$$

where $a = a1a0$ and $b = b1b0$.

It is quite intuitive that if two input vectors share a large number of bits, then the probability of both of them simultaneously satisfying the fault function is higher compared to the situation where vectors share fewer bits. In other words, if an input vector $v_i = a_i b_i$ (a_i and b_i are two operands concatenated together) activates a fault then, another input vector $v_j = a_j b_j$ which shares many bits with (or "looks very similar" to) v_i, would also very likely activate the same fault. Thus, we make the following observation:

The probability of an input vector activating the fault conditioned upon another vector activating the fault, referred to as 'conditional failure probability' (CFP), increases as the number of bits shared between the two vectors increases.

In the example above, if vector v_i activates $z0$ stuck-at 0 and v_j shares atleast k bits with v_i, then the CFP of v_j has been tabulated in the Table 2.4 as a function of k. In order to compute CFP, since we know the fault function, we make use of the fact that CFP is same as the probability that $a0$ and $b0$ are amongst the shared bits. When 4 bits are shared i.e., input vectors are identical, CFP is 1. When atleast 0 bits are shared which means v_j can be any input vector, CFP is same as the unconditional failure probability. Since, $Probability(a0 = 1) = Probability(b0 = 1) = \frac{1}{2}$, $Probability(h_{z0=0}(a, b) = 1) = \frac{1}{4}$.

k	0	1	2	3	4
CFP	0.25	0.27	0.37	0.60	1.00

Table 2.4: Conditional Failure Probability (CFP) as a function of the number of bits guranteed to be shared (k), for fault $z0$ stuck-at 0 in the 2-bit multiplier logic. As k increases, CFP increases exponentially. $k = 0$ implies atleast 0 bits are shared which means they can be any two 4-bit input vectors.

This observation is more strongly confirmed through verilog simulations on three different synthesized gate-level 8-bit multiplier designs. One of them is a general design synthesized using Cadence RTL Compiler [Cad] (D1) and the other two designs are synthesized using Synopsys Design Compiler [Syn], out of which one is a general design (D2) and another is based on "carry save array" synthesis model (D3) - a Synopsys DesignWare Building Block IP. For each design we injected faults in three different locations (L1, L2 and L3) and exhaustively simulated each design-location with all possible 2^{16} input vectors. Since the fault function is very difficult to deduce for such a large design, CFP is computed as the fraction of fault activating input vectors amongst all the input vectors that share *atleast* k bits with a fault activating input vector. Then, its averaged over all the fault activating input vectors. Fig.2.3 plots the average CFP as a function of k. CFP rises exponentially with the amount of bit sharing. $k = 0$ refers to "atleast zero bits shared" which means any and every input vector. Hence, the CFP at $k = 0$ is same as the unconditional failure probability. While bit sharing is a relative indicator of CFP, its absolute value is design dependent. For instance, when 12 bits are shared out-of 16, for design-location pair D1L1, CFP is 0.19 whereas for D3L3, CFP is 0.31.

In case of transient faults, correlation between CFP and number of bits shared is almost negligible because faults do not persist. Since fault at a location appears for a very short duration, even if $v_i = v_j$, one of them activating the fault does not

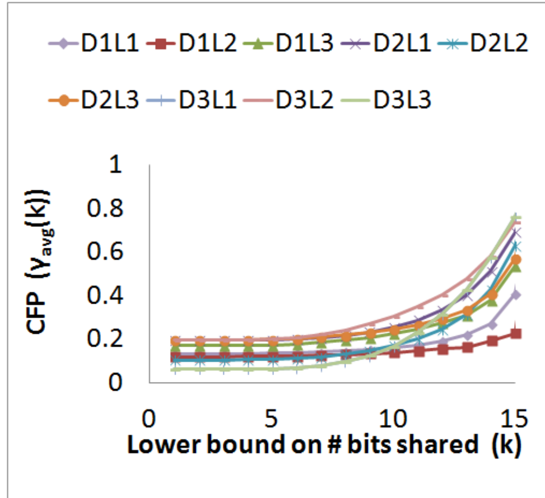

Figure 2.3: Average conditional failure probability (CFP, $\gamma_{avg}(k)$) as a function of the number of bits shared (k) between input vectors. Stuckat faults are injected in three different locations (L) of three different designs (D) of 8-bit multipliers.

necessitate activation by the other input vector. Intermittent faults - since they may persist for longer durations, exhibit positive correlation which is stronger than the transient faults but weaker than the permanent faults.

If the input vectors generated during an application run share a lot of bits amongst themselves, then their CFP is large implying that either a lot or a very few of them activate faults in a given run. In the next section, we'll derive variance in the fault rate as a function of CFP and define similarity to capture CFP. Our aim in this work is to curb the extreme fault rates by reducing CFP which is achieved by code transformations that reduce bit sharing.

2.1.2 Distinctive nature of permanent/intermittent delay fault model

Timing violations are contingent upon the sensitization of critical paths which in turn depends on the series of consecutive input vectors. Unlike stuck-at faults, knowledge of the current input vector alone is not sufficient to determine a timing violation. If the faulty unit is assumed to be isolated and not receiving any off

path inputs, then pair of the current and the previous input vectors to the faulty unit is sufficient to model the delay faults, as also argued in [ZKE12]. Therefore, if F is a delay fault, $v = ab$ is any input vector that accesses the faulty unit and $v' = a'b'$ is the input vector previous to v, then,

$$h_F(a, b, a', b') = 1 \Leftrightarrow \{a, b\}\ preceeded\ by\ \{a', b'\}\ activate\ the\ fault\ F$$

The activation of a delay fault can, thus, be determined by two input vectors - the current and the previous input vector. Fig.2.4 shows the average CFP associated with a pair of vectors that share atleast k bits with another fault activating pair of vectors. These results are obtained from gate-level timing simulations of D1, D2 and D3 designs back-annotated with standard delay format (SDF) file and operated at two different frequency overscaling factors - 10% (F1) and 5% (F2). Since exhaustive simulations would have required 2^{32} pairs of vectors, each vector being 16-bit wide, we perform 100 Monte-Carlo runs and in each run 66000 input vectors are simulated. Results shown in the figure are averaged over these 100 Monte-Carlo runs. Like in the case of stuck-at faults, here as well, the curve exponentially grows with sharing larger than half the bits ($k = 16$). For lower values, it remains almost flat. Also the absolute value of CFP for lower values of k is much smaller compared to the stuckat faults because due to small frequency overscaling factors very specific pairs of input vectors cause timing violations (or, activate a delay fault).

Hereonwards, the theory and the conclusions thereof shall be built upon the positive correlation between CFP and bit sharing. Since both the fault models show this correlation, we'll discuss only stuck-at fault model and, unless and otherwise mentioned, same conclusions shall be applicable to the delay faults as well, with the only difference that in the case of stuck-at faults, a single input vector is considered whereas in the case of delay faults, two consecutive input vectors are considered.

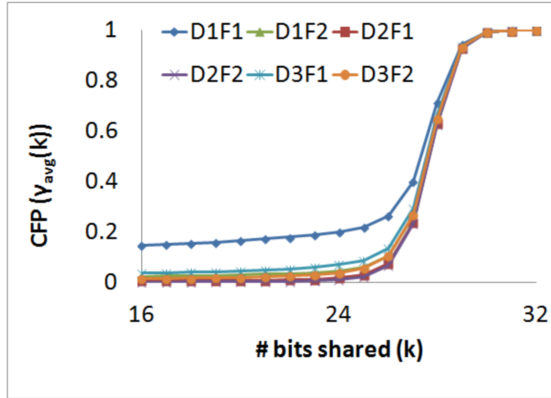

Figure 2.4: Average conditional failure probability (CFP, $\gamma_{avg}(k)$) as a function of the number of bits shared ($k > 15$) between two pairs of input vectors. For lower values of k, the curve almost remains flat.

App	Mult Count	Output Type	Correlation
Ac	42195	1-D int vector	0.80
Ft	45056	1-D complex int vector	0.12
Fr	45000	1-D int vector	0.80
Km	46000	A double precision number	0.03
Mm	39304	An int matrix	0.80

Table 2.5: Benchmark kernels: Count of multiply instructions, output type and correlation between fault rate and error magnitude as obtained from fault injection experiments with permanent stuck-at fault model.

2.2 Analytical Modeling of Fault Rate

In this section, we derive variance in the fault rate as a function of CFP and define similarity to capture CFP. Before doing the mathematical analysis we would like to motivate the relevance of fault rate as a metric.

2.2.1 Fault Rate as a Metric

In this work, we are experimenting with following 5 integer benchmark kernels: Autocorrelation (Ac), FIR Filter (Fr), Matrix Multiplication (Mm), Kmeans Clustering (Km) and FFT (Ft). While Ac, Fr and Mm are simple multiply-accumulate kernels where result of every multiply operation gets accumulated in the final output, Km and Ft are comparatively more complex applications where results of multiply operations can get masked with later operations in the code. We injected permanent stuck-at faults at several locations (one at a time, more details on the experimental setup are in the Chapter 4), and recorded the fault rate and the error magnitude from several runs. Table 2.5 shows the average correlation between these two quantities. We observe that for Ac, Fr and Mm, correlation is 0.80 but for Km and Ft correlation is very low (< 0.12). This is expected because latter applications have high application-level masking. Also shown in the table are the number of multiply instructions executed by each kernel in a single run and the type of final output. Error magnitude for Ac, Fr, Km and Ft is computed as the l_2 *norm* of the difference between the observed faulty output and the golden output. For Mm, we compute the "absolute sum value" norm of the residual matrix. Therefore, for kernels with low application-level masking, reducing fault rate reduces error magnitude as well.

Timing speculative architectures [EKD03] and their applications as in [SSK11] exploit the rarity of sensitization of the critical paths to operate at the underscaled voltages. With the help of hardware detection and recovery mechanisms, the rare timing violations are corrected with some penalty, but because in the common case timing is met, overall energy is saved. The energy benefits of such timing speculative architectures critically require low fault rate. Similarly, performance penalty due to recovery and reprocessing in checkpointing and roll back based fault tolerance mechanisms is directly proportional to the fault rate [TR84].

2.2.2 Analytical Model

Assume that an application executes N instructions which generate N input vectors feeding the faulty hardware. Let $\mathcal{X}_i$ indicate the event that a fault is activated by the ith input vector v_i. $\mathcal{X}_i$ is then a Bernoulli random variable:

$$\mathcal{X}_i = \begin{cases} 1 & p_i \\ 0 & 1 - p_i \end{cases}$$

Where p_i is the fault activation probability (referred henceforth as failure probability).

$$p_i = Probability(h_F(i) = 1)$$

Then we can express $\mathcal{FR}$ as,

$$\begin{aligned} \mathcal{FR} &= \frac{\sum_{i=1}^{N} \mathcal{X}_i}{N} \\ \mu_{\mathcal{FR}} &= \frac{1}{N}\left(E\left[\sum_{i=1}^{N} \mathcal{X}_i\right]\right) = \frac{1}{N}\left(\sum_{i=1}^{N} p_i\right) \\ \sigma^2_{\mathcal{FR}} &= \frac{1}{N^2}\left(E\left[\left(\sum_{i=1}^{N} \mathcal{X}_i\right)^2\right] - \left(\sum_{i=1}^{N} E\left[\mathcal{X}_i\right]\right)^2\right) \\ \omega_{\mathcal{FR}} &= \mu_{\mathcal{FR}} + 3 * \sigma_{\mathcal{FR}} \end{aligned}$$

$\mu_{\mathcal{FR}}$, $\sigma_{\mathcal{FR}}$ and $\omega_{\mathcal{FR}}$ are defined for a given fault F and application A, evaluated over all the inputs I.

Since permanent faults are activated by only certain input vectors, $\sigma^2_{\mathcal{FR}}$ can be decomposed into two factors:

1. Variance due to randomness in the input vector itself which is captured by the non-extreme values of p_i. p_i is a function of the distribution of input vector v_i, the implementation of the faulty hardware and location of the fault, if it is a stuck-at fault. In case of delay faults, critical paths depend on the specific implementation itself.

2. Variance due to correlation between the input vectors. Since two input vectors v_i and v_j can be highly correlated, for instance, they can be identical, events $\mathcal{X}_i$ and $\mathcal{X}_j$ may also be correlated. Mathematically, $p_{j|i} = Probability(\mathcal{X}_j = 1|\mathcal{X}_i = 1)$ need not be same as p_j. $p_{j|i}$ is the CFP discussed in the Section 2.1. Since $p_{i|j}$ and $p_{j|i}$ may not be same, CFP is redefined as, $\gamma_{ij} = \frac{p_{i|j}+p_{j|i}}{2}$. Hence, $\gamma_{ij} = \gamma_{ji}$.

$\sigma^2_{\mathcal{FR}}$ in terms of p_i and CFP can be derived as follows:

$$\begin{aligned}
\sigma^2_{\mathcal{FR}} &= \frac{1}{N^2}\left(E\left[\left(\sum_{i=1}^{N}\mathcal{X}_i\right)^2\right] - \left(\sum_{i=1}^{N}E\left[\mathcal{X}_i\right]\right)^2\right) \\
&= \frac{1}{N^2}\left(E\left[\sum_{i=1}^{N}X_i^2\right] + E\left[\sum_{i=1}^{N}\sum_{j=1,j\neq i}^{N}X_i.X_j\right] - \left(\sum_{i=1}^{N}p_i\right)^2\right) \\
&= \frac{1}{N^2}\left(\sum_{i=1}^{N}p_i - \sum_{i=1}^{N}(p_i)^2 - 2\sum_{i=1}^{N}\sum_{j=1,j>i}^{N}p_ip_j + E\left[\sum_{i=1}^{N}\sum_{j=1,j\neq i}^{N}X_iX_j\right]\right)
\end{aligned} \tag{2.1}$$

where $E\left[\sum_{i=0}^{N}\sum_{j=1,j\neq i}^{N}X_iX_j\right]$ can be computed as follows:

$$\begin{aligned}
E\left[\sum_{i=0}^{N}\sum_{j=1,j\neq i}^{N}X_iX_j\right] &= \sum_{i=1}^{N}\sum_{j=1,j>i}^{N}\left(E\left[X_iX_j\right] + E\left[X_iX_j\right]\right) \\
&= \sum_{i=1}^{N}\sum_{j=1,j>i}^{N}\left(p_ip_{j|i} + p_jp_{i|j}\right)
\end{aligned}$$

Therefore,

$$\sigma^2_{\mathcal{FR}} = \frac{1}{N^2}\underbrace{\left(\sum_{i=1}^{N}\left(p_i - p_i^2\right)\right)}_{\text{Randomness due to individual instruction}} + \frac{1}{N^2}\left(\underbrace{\sum_{i=1}^{N}\sum_{j=1,i>j}^{N}\left(p_ip_{j|i} + p_jp_{i|j} - 2p_ip_j\right)}_{\text{Randomness due to correlation}}\right) \tag{2.2}$$

Thus the Eqn.(2.2) shows $\sigma^2_{\mathcal{FR}}$ as a function of CFP. So far we have established that the $\sigma^2_{\mathcal{FR}}$ is a function of CFP which is proportional with the amount of bit sharing. Therefore, certain applications whose input vectors share large number of bits, are susceptible to large fault rates. Since the amount of bit sharing is a property of the code, partly independent of the input, we motivate a code metric called *Similarity* (S), to capture the impact of bit sharing, in other words, CFP. Its defined as follows:

$$
\begin{aligned}
S &= \frac{1}{N^2}\sum_{i=1}^{N}\sum_{j=1,i>j}^{N}\left(p_{j|i} + p_{i|j} - p_i - p_j\right) \\
&= \frac{1}{N^2}\sum_{i=1}^{N}\sum_{j=1,i>j}^{N}\left(2\gamma_{ij} - p_i - p_j\right)
\end{aligned} \tag{2.3}
$$

Subtracting p_i and p_j emphasize the contribution of correlation between input vectors towards $\sigma^2_{\mathcal{FR}}$. If all the input vectors are uncorrelated then $S = 0$. In other words, similarity does not contribute towards $\sigma^2_{\mathcal{FR}}$. Assuming, $p_i = p \ \forall \ i$, pS is the second term in the Eqn.(2.2).

Due to impersistent nature of the transient faults, fault activation by a given input vector usually does not influence the fault activation by another input vector. Hence, for the transient faults, $p_{j|i} \sim p_j$ and as a result, as $N \to \infty$, $\sigma^2_{\mathcal{FR}} \to 0$. In case of intermittent faults, γ_{ij} not only depends on the correlation between the input vectors but also the distance between them. If two input vectors are identical but separated by large number of instructions (more than the duration of the fault), even if one of them activates the fault, other would not. Consequently, the contribution of the correlation term in the Eqn.(2.2) reduces with reduction in the fault duration.

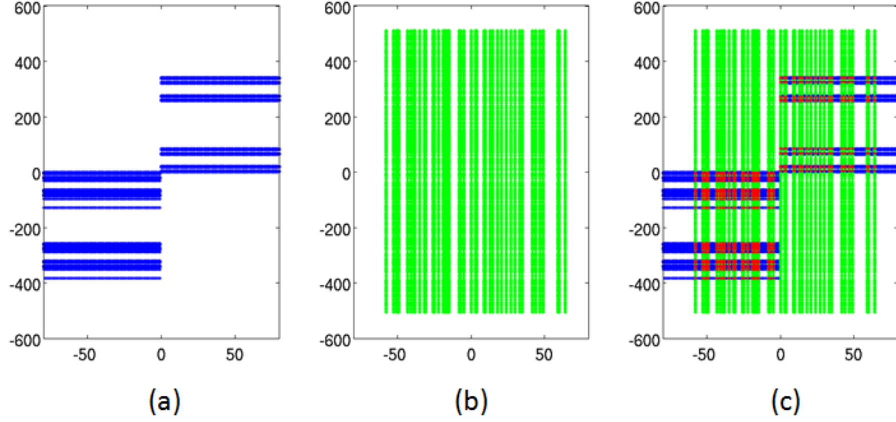

Figure 2.5: (a) Subset of fault activating input vectors (b) Input vectors generated by a faulty run of Fr application (c) overlap between the (a) and (b) (shown in red).

2.3 Pictorial Representation of Fault Rate

Every input vector is composed of two operands and hence, it corresponds to a unique point in the 2-D integer space. Therefore, fault rate can be visualized as the overlap between the input vectors generated by the application and the set of fault activating input vectors. Fig.2.5a-c show the input vectors generated by application Fr in a faulty run, subset of input vectors that activate the fault and the overlap between the two. The amount of overlap is proportional to the fault rate.

2.4 Chapter Summary

In this chapter, we have discussed the fault models that we use to model the hardware failure mechanisms. Utilizing the fault model characteristics, we make an observation that correlates the CFP for a pair of input vectors with the number of bits shared between them. Therefore, large amounts of bit sharing amongst the input vectors generated during a faulty run of an application implies large CFP which in turn implies that either a lot or a very few of them simultaneously

activate the fault (high $\sigma^2_{\mathcal{FR}}$). Thus, in order to formally express $\sigma^2_{\mathcal{FR}}$ as a function of CFP, we analytically model the fault rate as a random variable. Since we are interested in curbing the high fault rates observed for certain applications, we define a code metric called *Similarity* that captures the CFP. In the next chapter, we shall propose a practical approximation to statically compute similarity and also discuss code transformations to reduce it.

CHAPTER 3

Similarity and Code Transformations

In this chapter we propose an approximate definition/procedure to statically estimate similarity based on the observation that CFP is directly proportional to the number of bits shared. Then we present a couple of example code transformations that reduce similarity and thereby, reduce variance and the worst case fault rate.

3.1 Practical Approximation to Similarity

Similarity is defined in the Section 2.2.2, as

$$S = \frac{1}{N^2} \sum_{i=1}^{N} \sum_{j=1, i>j}^{N} (2 * \gamma_{ji} - p_i - p_j)$$

where p_i and p_j are the unconditional failure probabilities of input vectors v_i and v_j and γ_{ji} is the CFP. From the discussion in the Section 2.1 we had empirically as well as intuitively established that CFP is proportional to the number of bits shared between the two input vectors. Since the amount of bit sharing is a good relative indicator of the CFP, that can approximate similarity for the code. But this requires thorough profiling and can not be done statically at compile time.

S	Similarity
S_r	Relaxed similarity
S_s	Strict similarity

Table 3.1: Notations introduced in this chapter

Org	Original code
Sw	"Swap" transformation
SwN	"Swap-Negate" transformation

Table 3.2: Acronyms introduced in this chapter

However, using this observation, we propose an approximate definition/procedure to efficiently estimate the similarity:

$$S \sim \frac{1}{N^2} \sum_{i=1}^{N} \sum_{j=1, i \neq j}^{N} w_{ij} \tag{3.1}$$

where, w_{ij} is a weight assigned to a pair of input vectors $\{v_i, v_j\}$ in proportion to the number of "non-constant" operands shared between them. For simplicity, we equate w_{ij} to the number of non-constant operands shared. But the actual value of the weights would depend on the design and the fault location. 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 the following. Assume v_i and v_j are two input vectors in stuck-at fault model. Let $\{a_i, b_i\}$ and $\{a_j, b_j\}$ be their operands, respectively, such that $b_i = b_j = K$ and a_i and a_j share atleast 0 bits, in other words, they are independent. Then,

$$\begin{aligned}
\gamma_{ij} &= Probability(\mathcal{X}_i = 1 | \mathcal{X}_j = 1) \\
&= Probability(h_F(a_i, K) = 1 | h_F(a_j, K) = 1) \\
&= Probability(h_{F,K}(a_i) = 1 | h_{F,K}(a_j) = 1) \\
&= Probability(h_{F,K}(a_i) = 1) \text{ since, } a_i \text{ and } a_j \text{ are independent} \\
&= Probability(h_F(a_i, K) = 1) \\
&= p_i
\end{aligned}$$

where, $h_{F,K}(a_i) := h_F(a_i, K)$. Hence, such input vectors do not contribute towards the similarity as per Eqn.(2.3). Based on the results of verilog simulation on different 8-bit multiplier designs for location L1 and frequency overscaling factor (F1=10%) (see Section 2.1), we computed CFP as a function of the number of operands shared. The results are shown in the Fig.3.1a and 3.1b, respectively. There are following observations:

- With increase in the number of operands shared, CFP also increases. This is expected because in case of stuck-at faults, atleast 1 operand sharing implies either half or all the bits are shared in the extreme cases and three-fourth bits are shared in the average case.
- Operand sharing is only a relative indicator of the CFP. Its absolute value is design dependent. Therefore, for a single operand sharing, while CFP for D1L1 is 0.25, for D2L1 it is 0.30. This is even more evident for delay faults.
- While in case of stuck-at faults, two operand sharing guarantees a CFP of 1, in case of delay faults, four operand sharing results in a CFP of 1. This is because delay fault activation is dependent on the two consecutive input vectors.

Similarity computation under stuck-at fault model Vs delay fault model: Similarity computed under stuck-at fault model considering operand sharing amongst pairs of single input vectors is almost half the similarity computed under delay fault model where operand sharing amongst pairs of two consecutive input vectors are considered. This is so because *for every pair* of input vectors sharing atleast one operand, there are *two pairs* of consecutive input vectors sharing that many operands. An example is shown in the Fig.3.2. Rarely observed three or more operand sharing makes stuck-at fault similarity slightly less than half of the delay fault similarity.

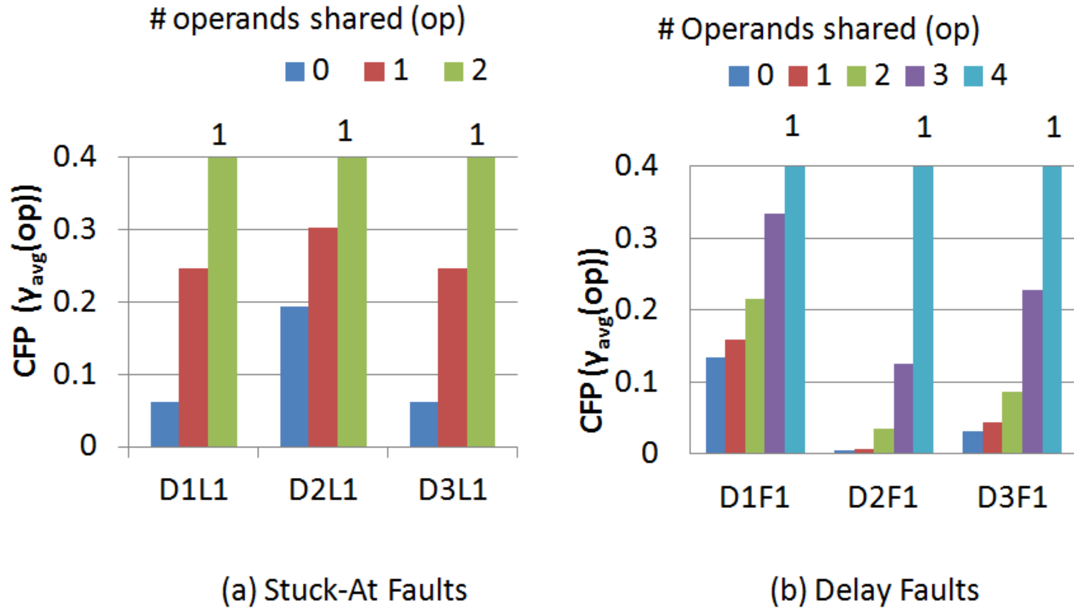

Figure 3.1: CFP as a function of the number of operands shared (*op*), as obtained from verilog simulations on 8-bit multiplier designs (see Section 2.1).

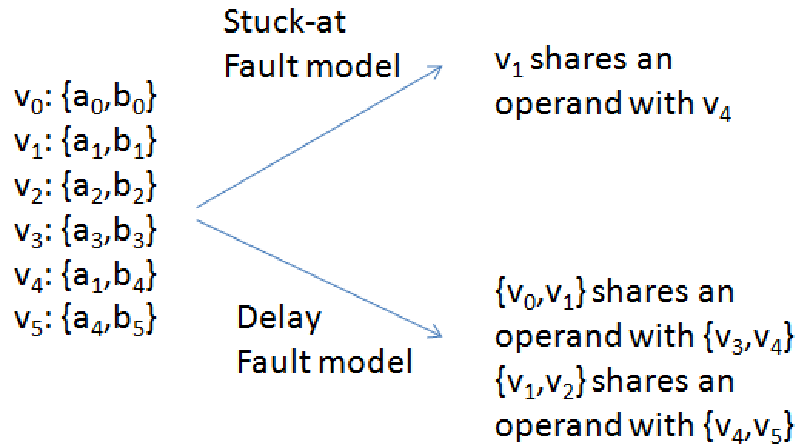

Figure 3.2: An operand being shared between two input vectors, namely v_1 and v_4 implies that there exist two pairs of two consecutive input vectors, namely, $\{\{v_0, v_1\}\{v_3, v_4\}\}$ and $\{\{v_1, v_2\}\{v_4, v_5\}\}$, each sharing an operand.

Strict vs relaxed operand sharing: In stuck-at fault model, we distinguish between two types of operand sharing: *strict* and *relaxed*. Strict operand sharing occurs between two input vectors v_i and v_j when $(a_i == a_j)||(b_i == b_j)$ and relaxed operand sharing occurs when $(a_i == a_j)||(b_i == b_j)||(a_i == b_j)||(b_i == a_j)$. In the discussion above, operand sharing was strict. Corresponding to these two types of operand sharing, we can compute *strict similarity* (S_s) and *relaxed similarity* (S_r), where the latter is always greater than or equal to the former. This distinction is necessary because for certain faults that satisfy $h_F(a, b) = h_F(b, a)$, reduction in strict similarity, say, by swapping the operands, does not guarantee reduction in $\sigma_{\mathcal{FR}}$ because activation of such faults is independent of the operand order. For instance, consider $z0$ stuck-at 0, in Fig.3.3. Its fault function is $h_{z0=0}(a, b) = a0b0 = h_{z0=0}(b, a)$. Therefore, in a set of vectors that share one of the operands, by swapping the operands of a few of them, strict similarity can be reduced but fault rate would not be affected. To rectify this anamoly, relaxed operand sharing should be considered because it is also immune to operand order. On the other hand, for other types of faults, $\sigma_{\mathcal{FR}}$ may reduce while relaxed similarity does not change. An example for such a fault is n_1 stuck-at 0 in the same figure. Its fault function is $h_{n_1=0}(a, b) = a0b1 \neq b0a1 = h_{n_1=0}(b, a)$. Hence, by swapping the operands, relaxed similarity would not change but fault rate would change. Therefore, strict operand sharing needs to be considered for such types of faults in order to correlate similarity with the $\sigma_{\mathcal{FR}}$. In practice, its difficult to determine if a fault function is commutative with respect to the operand order, we'll propose a code transformation to reduce both - relaxed as well as the strict similarity. In delay fault model, operand order almost always matters due to transistor stacking effect.

Static vs profile-based similarity computation. Using the procedure outlined in Eqn.(3.1), strict as well as relaxed similarity which is inherent to a code, can be statically computed. An example computation of strict similarity for stuck-

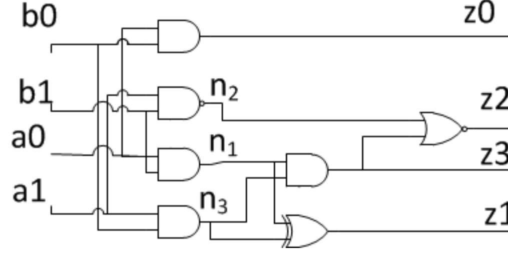

Figure 3.3: A general 2-bit multiplier logic block

```
for ( i = 0; i < N; i++ ) {
    d[i] = e * f[i]
}
```

Figure 3.4: An example code in its original version.

at fault input vectors is the following: consider the computation kernel code shown in Fig.3.4. N input vectors share the first operand with the value e. Thus, there are $\left(\frac{N(N-1)}{2}\right)$ pairs strictly sharing an operand. In absence of any information about $f[i]$ and e, operand sharing amongst elements of array f or amongst element of array f and scalar e can not be ascertained. Therefore, for this code, $S_r = S_s = \frac{N-1}{2N}$. 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 profiled executions needs to be computed. Since one of our benchmark kernels, Km, has zero compiler visible operand sharing amongst multiply instructions, we compute similarity using fault-free profiled runs.

3.2 Code Transformation

Since applications with large amounts of operand sharing are more susceptible to high fault rates, it is in the best interest to reduce the operand sharing. In this section, we present two simple architecture independent code transformations to

```
for ( i = 0; i < N; i=i+2 ) {
    d[i] = e * f[i];
}

for ( i = 1; i < N; i=i+2 ) {
    d[i] = f[i] * e;
}
```

Figure 3.5: Code of Fig.3.4 after applying transformation *Swap* (Sw). Half the operands are swapped.

reduce similarity and thereby, curb the worst case fault rates, $\omega_{\mathcal{FR}}$.

First transformation is called as *Swap* (Sw). In Sw, for-loop is divided into equal halves and each half increments by 2 instead of 1. While the operand order is unchanged in one of the loops, in the other loop, operands are swapped. Fig.3.5 shows the application of Sw on the example original code (Org) of the Fig.3.4. As computed in the previous section, $S_{s,Org} = \frac{N-1}{2N}$ due to N input vectors sharing an operand. However, with Sw, there are only half as many input vectors sharing a particular operand, although there are two sets of them - one sharing the first operand and the other sharing the second operand. Therefore, when statically computed, strict similarity would be,

$$S_{s,Sw} = \frac{1}{N^2}\left(2\left[\frac{\frac{N}{2}\left(\frac{N}{2}-1\right)}{2}\right]\right) = \frac{N-2}{4N}$$

which is approximately, 2X smaller than $S_{s,Org}$. Relaxed similarity does not change, however. For Km, both the operands of all the multiplies are same due to the nature of distance computation kernel - $(x * x + y * y)$. Therefore, for Km, Sw leaves Org unchanged.

Second transformation, *Swap-Negate* (SwN), is an improvement over Sw to reduce even the relaxed similarity. In SwN, operands are not only swapped but

```
for ( i = 0; i < N; i=i+2 ) {
    d[i] = e * f[i];
}
t2 = -e;
for ( i = 1; i < N; i=i+2 ) {
    d[i] = (-f[i]) * t2;
}
```

Figure 3.6: Code of Fig.3.4 after applying transformation *Swap-Negate* (SwN). Half the operands are swapped and multiplied by -1.

also multiplied by -1. Reduction in strict similarity is obvious, as discussed above. Reduction in the relaxed similarity comes about because the value of the operand being shared by the input vectors in one of the sets is e, but in the other set its $-e$. Therefore, with SwN both the relaxed as well as the strict similarity are reduced. Fig.3.6 shows the application of SwN to Org. In presence of loop-carried dependences, the two for-loops should be interleaved to maintain the iteration order.

App	S_r(1e-3)			S_s(1e-3)		
	Org	Sw	SwN	Org	Sw	SwN
Fr	15.9	15.9 (1.00)	10.5 (0.66)	14.9	7.95 (0.53)	7.94 (0.53)
Ac	8.70	8.70 (1.00)	5.32 (0.61)	5.50	4.36 (0.79)	3.25 (0.59)
Mm	8.60	8.60 (1.00)	8.17 (0.95)	4.70	4.31 (0.92)	4.30 (0.91)
Km	0.40	0.40 (1.00)	0.37 (0.96)	0.40	0.38 (0.96)	0.37 (0.96)
Ft	0.10	0.10 (1.00)	0.10 (0.98)	0.04	0.04 (1.00)	0.04 (1.00)

Table 3.3: Relaxed (S_r) and strict (S_s) similarity values for the original (Org), swapped (Sw) and swap-negated (SwN) codes of all the applications. Similarity values normalized with respect to Org are shown in the brackets.

Table 3.3 tabulates the relaxed and strict similarity under stuck-at fault model due to both the code transformations and the original code. Following are the important observations:

- Relaxed similarity for a given code version is always greater than the strict similarity. This is expected because strict operand sharing is a special case of relaxed operand sharing.
- Sw does not reduce the relaxed similarity whereas SwN reduces both the similarity values. This has already been explained above.
- In all but one application, namely Ac, reduction in strict similarity due to either code transformations is almost same because both of them swap the operands and thereby reduce strict operand sharing.

Ft and Km have very low absolute similarity and hence, much less scope for reducing it further.

Sw and SwN are two different mechansims to recursively break a loop into two independent loops and swap the operands in one of the loops. Additionally, in case of SwN, swapped operands are negated. Thus, they can be applied to any arithematic operation, not only multiplies, irrespective of operand sharing. However, when SwN is applied to adder operations final result of the add operation should be negated for correctness. Moreover, whenever either of them are applied where there is no operand sharing, transformed S and $\sigma_{\mathcal{FR}}$ are dictated by the inputs to the application and hence, the reduction is not guaranteed. We'll see in the results that for Fr, Ac and Mm, transformations on adder operations can actually hurt because none of them have any compiler visible operand sharing and their absolute similarity values are very low.

Code transformations "spread out" the application's input vectors. Fig.3.7a-c show the input vectors generated by the Org, Sw and SwN versions of Fr in a

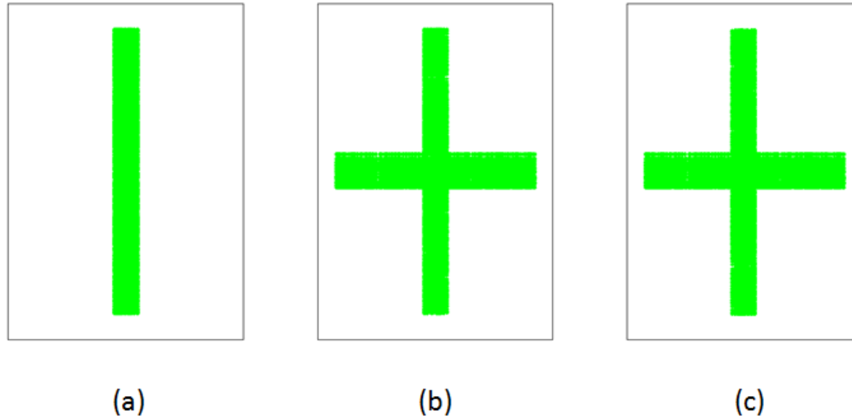

Figure 3.7: Input vectors generated in a faulty run of Fr's original (a) and transformed codes - Sw (b) and SwN (c).

faulty run. Swapping is equivalent to mirroring across $x = y$ line and negating is equivalent to mirroring across the origin. Since both the transformations swap the operands and input vectors in the Org *look* symmetric across the origin, plots (b) and (c) *look* identical.

3.3 Chapter Summary

In this chapter we've proposed an approximate definition for the similarity metric based on operand sharing and outlined a procedure to statically compute it. Since applications with larger similarity values are more susceptible to larger worst case fault rates, we proposed two simple and architecture independent code transformations to reduce similarity. In the next chapter, we'll present the fault injection infrastructure and discuss results of fault injection experiments, particularly the impact of code transformations on variance in the fault rate and the worst case fault rate.

CHAPTER 4

Experimental Setup and Results

In this chapter, we'll describe the experimental set up including fault injection infrastructure and fault inject parameters, and discuss the results of fault injection experiments.

4.1 Experimental Setup

4.1.1 Hierarchical Fault Simulation

The common mode to evaluate any fault tolerance mechanism has been the injection of specific fault models like stuck-at, random bit-flip etc. at various design abstractions like gate-level, micro- architecture level etc. Both are critical factors in the evaluation methodology. Since fault models have already been discussed in Chapter 2, here we'll focus on the second factor, namely, the abtraction level at which faults are injected. In our work, we want to study the impact of faults modeled at the gate-level on applications. Due to several levels of hierarchy in between, although accurate, full gate-level simulation is very slow. On the other hand, fault injection at micro- architecture or higher levels does not capture the complex interactions in the lower levels. For instance, a stuck-at fault injected at micro-architecture level can be implemented by fixing an output bit of a latch to 0 or 1 but in reality a stuck-at fault at gate level might affect several output bits [LRK09]. To deal with this trade-off between speed and accuracy, in the past, hierarchical simulation infrastructures have been proposed [LRK09] [MLN02] [KIR99]

t_a	Duration of active cycle for intermittent faults
t_i	Duration of idle cycle for intermittent faults
l_{burst}	Burst length - number of active-idle cycles
$\mathcal{FR}$	Fault rate random variable
$\mathcal{EM}$	Error magnitude random variable
$\mu_{\mathcal{FR}}$	Fault rate for a given F and A, averaged over all I
$\sigma_{\mathcal{FR}}$	Standard deviation in fault rate for a given F and A, computed over all I
$\omega_{\mathcal{FR}}$	Worst case fault rate defined as $\mu_{\mathcal{FR}} + 3 * \sigma_{\mathcal{FR}}$
$\omega_{\mathcal{EM}}$	Worst case error magnitude
$\mu_{\mathcal{FR}}^{norm,avg}$	$\mu_{\mathcal{FR}}$ normalized and averaged over F
$\sigma_{\mathcal{FR}}^{norm,avg}$	$\sigma_{\mathcal{FR}}$ normalized and averaged over F
$\omega_{\mathcal{FR}}^{norm,avg}$	$\omega_{\mathcal{FR}}$ normalized and averaged over F
$\omega_{\mathcal{EM}}^{norm,avg}$	$\omega_{\mathcal{EM}}$ normalized and averaged over F
S_s	Strict similarity

Table 4.1: Frequently used notations

CFP	Conditional Failure Probability
Fr, Ac, Mm, Km, Ft	Benchmark kernels: Fir, Autocorrelation, Matrix Multiplication, Kmeans clustering, FFT
Org	Original code
Sw	“Swap” transformation
SwN	“Swap-Negate” transformation
PS, IS, PD	Fault models: Permanent Stuck-at, Intermittent Stuck-at, Permanent Delay

Table 4.2: Frequently used acronyms

[PBM00]. To improve the performance of gate-level fault simulations, Mirkhani *et. al.* combine behavioral and gate-level VHDL models. While gate-level models are used to inject and propagate faults in the faulty hardware, behavioral models are used to propagate faults that appear at the input ports. Kalbarczyk *et. al.* prepare fault dictionaries for a give nfault model through off-line lower- level simulations and use them for fault propagation during higher level simulation. Li *et. al.* couple a microarchitectural simulator with a gate-level timing level simulator which is selectively and on-demand invoked to perform accurate gate-level fault simulations.

Our fault injection infrastructure closely resembles [LRK09] as far as delay fault simulations are concerned. We inject faults in architecture visible multiply instructions using VarEmu [rev], an instruction-level emulator. A gate-level timing simulator, Mentor ModelSim, back-annotated with the 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. Communication between ModelSim and VarEmu is implemented via linux sockets. 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. Since implementating a timing model in C++ is extremely difficult, similar translation for delay fault simulation was not possible.

4.1.2 VarEmu

Fig.4.1a presents an overview of the VarEmu architecture for fault model emulation in our experiment. Applications in a virtual machine (VM) interact with VarEmu by enabling and disabling faults in the execution of emulated instructions. An operating system driver handles the interaction of applications with a virtual

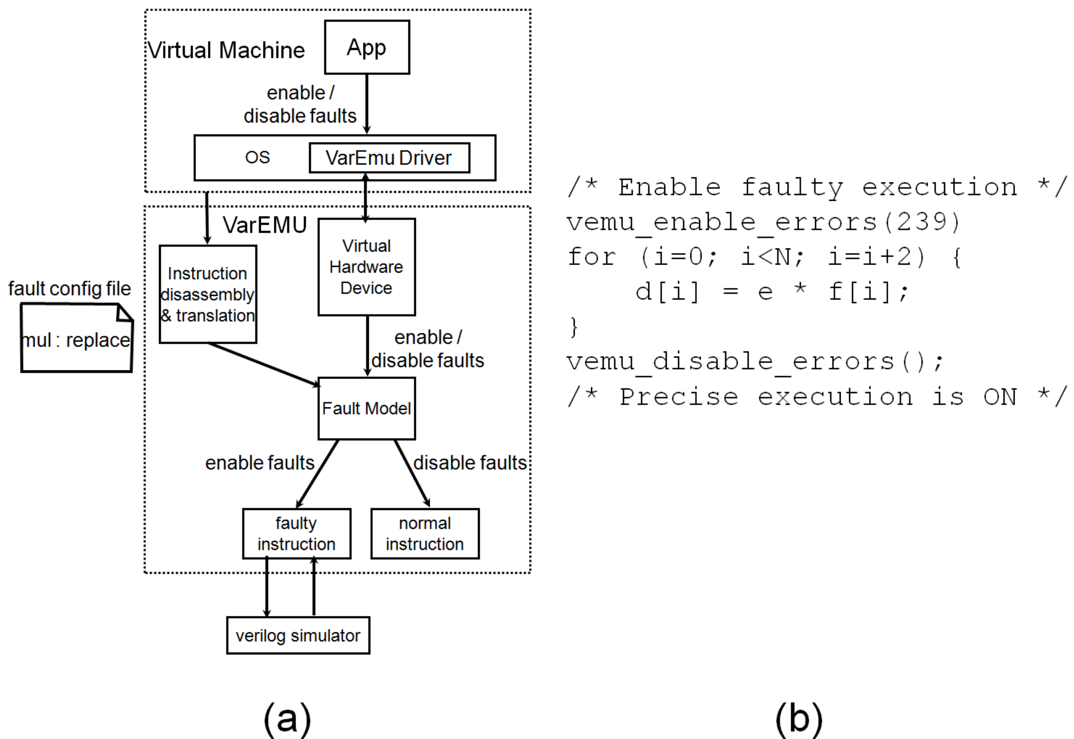

Figure 4.1: (a) VarEmu architecture for fault model emulation. (b) Enabling faults using VarEmu based fault model implementation

hardware device which exposes the VarEmu interface to the VM. When starting VarEmu, user provides a configuration file that contains instructions selected by the user to be configured as faulty instructions. During execution of the translated code, the decoded instruction contains a parameter which defines whether this instruction is susceptible to faults or not. The faulty instruction is replaced by an alternative function. Inside that function, for delay faults, operands of the instruction are communicated to the ModelSim which simulates the design and returns the output. Enabling and disabling faults are done using a fine-grained switch of calling a system-call function in the emulated software, as illustrated in Fig.4.1b. OS has been kept fault-free.

4.1.3 Fault Injection Parameters

For fault injection experiments we use Cadence RTL Compiler synthesized 32-bit multiplier design. We inject three fault models: Permanent/Stuck-at (PS), Permanent/Delay (PD) and Intermittent/Stuck-at (IS), in the computation kernel of the application. Since we want to study the impact of hardware faults on SDC, we do not inject faults in OS which often has some detectable catastrophic impact.

For fault types PS and IS, 300 candidate fault locations are randomly chosen and a fault - stuck-at (0 or 1), is injected in one of those locations. While, for the PS model, fault remains alive through out the execution of the computation kernel, for the IS model, faults are injected in the beginning of the kernel. With t_a and t_i set to 100, we study the impact of three different burst lengths $l_{burst} = \{50, 500, 2500\}$ on correlation between the similarity and the variance in the fault rate. For fault type PD, frequency was overscaled by 20% to induce delay faults.

With 5 applications, each but Km[1] with three versions - one original and two transformed, and 100 randomly generated inputs per application version, there are $300*(4*3+1*2)*100 = 420,000$ faulty runs for PS model; $(300*(4*3+1*2)*100)*3 = 1,260,000$ faulty runs for the IS model and $1*(4*3+1*2)*100 = 1400$ faulty runs for the PD model. Each application version is compiled with ARM C/C++ compiler without any optimization flags. We'll separately study the impact of optimization flags on the efficacy of code transformations. We'll also briefly discuss the results of fault injection in adder instructions.

4.2 Results

In this section, we discuss the results of fault injection experiments, specifically, the average impact of the code transformations on $\sigma_{\mathcal{FR}}$, $(\sigma_{\mathcal{FR}}/\mu_{\mathcal{FR}})$, $\omega_{\mathcal{FR}}$ and $\omega_{\mathcal{EM}}$, relative (normalized) to the original code. $\mathcal{EM}$ is the error magnitude. To

[1] kmeans application has two versions - original and SwN transformed code

report these average normalized values for each one of the quantites we introduce four notations - $\sigma_{\mathcal{FR}}^{norm,avg}$, $(\sigma_{\mathcal{FR}}/\mu_{\mathcal{FR}})^{norm,avg}$, $\omega_{\mathcal{FR}}^{norm,avg}$ and $\omega_{\mathcal{EM}}^{norm,avg}$, respectively. For each fault-model/application pair, all four average normalized values have been computed in three steps: 1) We record $\mathcal{FR}$ and $\mathcal{EM}$ for every faulty run $\{A, I, F\}$. 2) Then we compute $\sigma_{\mathcal{FR}}$, $\sigma_{\mathcal{FR}}/\mu_{\mathcal{FR}}$, $\omega_{\mathcal{FR}}$ and $\omega_{\mathcal{EM}}$ over all Is. 3) All four quantities corresponding to the transformed codes are normalized with respect to the original code, and then averaged (arithmetic mean) over all those faults whose $\mu_{\mathcal{FR}} < 0.1$. We do not analyze faults with average fault rate more than 0.1 assuming that the higher fault rates would inevitably cause detectable catastrophic failures.

4.2.1 Impact of code transformations on $\sigma_{\mathcal{FR}}$ and $(\sigma_{\mathcal{FR}}/\mu_{\mathcal{FR}})$.

According to the Eqn.(2.2), $\sigma_{\mathcal{FR}}$ depends on failure probabilities, in addition to CFP. Thus, if failure probabilities which are roughly captured by $\mu_{\mathcal{FR}}$, dramatically change, reduction in similarity may not guarantee a reduction in $\sigma_{\mathcal{FR}}$. That's why, to be conservative and avoid interference due to drastic changes in $\mu_{\mathcal{FR}}$, we perform averaging and normalization only over those faults whose average fault rate due to the transformations remains within 1.25X and 0.75X of the original code. This is in addition to the above-mentioned upper bound of 0.1 on the average fault rate. Both these "filters" are mutually exclusive and serve entirely different purposes. We report $(\sigma_{\mathcal{FR}}/\mu_{\mathcal{FR}})^{norm,avg}$ for the same reason. Tables 4.3-4.5 show $\sigma_{\mathcal{FR}}^{norm,avg}$ and $(\sigma_{\mathcal{FR}}/\mu_{\mathcal{FR}})^{norm,avg}$ for PS, IS and PD fault models, respectively.

For PS faults (see Table 4.3), there are two important observations:

- Both the transformations reduce $\sigma_{\mathcal{FR}}$ as well as $\sigma_{\mathcal{FR}}/\mu_{\mathcal{FR}}$. Reduction in both these quantities follows the reduction in the strict similarity, S_s. While reduction is maximum for Fr (S_s reduces to 0.53X and $\sigma_{\mathcal{FR}}$ reduces to 0.57X

App	S_s		$\sigma_{\mathcal{FR}}^{norm,avg}$		$(\sigma_{\mathcal{FR}}/\mu_{\mathcal{FR}})^{norm,avg}$	
	Sw	SwN	Sw	SwN	Sw	SwN
Fr	0.53	0.53	0.58	0.57	0.61	0.61
Ac	0.79	0.59	0.83	0.79	0.82	0.77
Mm	0.92	0.91	0.72	0.69	0.72	0.69
Km	0.96	0.96	1.00	0.97	1.00	0.97
Ft	1.00	1.00	1.00	0.98	1.00	0.96

Table 4.3: Permanent/Stuck-at fault model: Correlating reduction in similarity (S_s) and reduction in $\sigma_{\mathcal{FR}}$ and $\sigma_{\mathcal{FR}}/\mu_{\mathcal{FR}}$ due to Sw and SwN transformations. Similarity values are normalized with respect to the original code.

by SwN), it is negligible for Km and Ft.

- Reduction in $\sigma_{\mathcal{FR}}$ as well as $\sigma_{\mathcal{FR}}/\mu_{\mathcal{FR}}$, due to SwN is consistently atleast as much as due to Sw. This is because, while Sw reduces only the strict similarity, SwN reduces both - the strict as well as the relaxed similarity.

For IS fault model, we experimented with three different burst lengths (l_{burst}) - $\{50, 500, 2500\}$. Corresponding fault models are referred to as IS0, IS1 and IS2, respectively. Table 4.4 reports average reduction in $\sigma_{\mathcal{FR}}$ and $\sigma_{\mathcal{FR}}/\mu_{\mathcal{FR}}$ due to SwN transformation. Impact of Sw transformation is very similar to SwN. Since the values for $\sigma_{\mathcal{FR}}$ and $\sigma_{\mathcal{FR}}/\mu_{\mathcal{FR}}$ are considerably different, we would analyze $\sigma_{\mathcal{FR}}/\mu_{\mathcal{FR}}$. As briefly discussed in the section 2.2.2, contribution of similarity to $\sigma_{\mathcal{FR}}$, depends on the intermittent fault duration. There are three observations:

- For less enduring faults, even significant reduction in similarity may not effect any reduction in $\sigma_{\mathcal{FR}}$. Therefore we observe in the Table 4.4, with IS0 fault model, while SwN reduces $\sigma_{\mathcal{FR}}/\mu_{\mathcal{FR}}$ for Ac to 0.78X, it increases to 1.17X for Mm.

- As fault duration increases, $\sigma_{\mathcal{FR}}/\mu_{\mathcal{FR}}$ reduces for all but Km.

App	$\sigma_{\mathcal{FR}}^{norm,avg}$			$(\sigma_{\mathcal{FR}}/\mu_{\mathcal{FR}})^{norm,avg}$		
	IS0	IS1	IS2	IS0	IS1	IS2
Fr	1.05	0.75	0.61	0.96	0.76	0.62
Ac	0.72	0.81	0.77	0.78	0.92	0.89
Mm	1.15	1.28	0.94	1.17	1.16	0.91
Km	0.80	1.10	1.10	1.30	1.11	1.07
Ft	0.93	1.31	1.07	0.88	1.04	0.97

Table 4.4: Intermittent/Stuck-at fault model: Table shows average normalized values of $\sigma_{\mathcal{FR}}$ and $\sigma_{\mathcal{FR}}/\mu_{\mathcal{FR}}$ due to SwN transformation for three different burst lengths ($l_{burst} = 50, 500, 2500$), corresponding to IS0, IS1 and IS2, respectively. Values are normalized with respect to the original code values corresponding to the respective burst length.

App	RC synthesized design				DC synthesized design			
	$\sigma_{\mathcal{FR}}^{norm,avg}$		$(\sigma_{\mathcal{FR}}/\mu_{\mathcal{FR}})^{norm,avg}$		$\sigma_{\mathcal{FR}}^{norm,avg}$		$(\sigma_{\mathcal{FR}}/\mu_{\mathcal{FR}})^{norm,avg}$	
	Sw	SwN	Sw	SwN	Sw	SwN	Sw	SwN
Fr	0.62	0.53	0.96	0.84	0.02	0.02	0.24	0.27
Ac	1.01	0.98	0.99	1.01	0.75	0.65	0.86	0.78
Mm	0.66	0.67	0.67	0.68	0.76	0.89	0.75	0.89
Km	NTV	NTV	NTV	NTV	NTV	NTV	NTV	NTV
Ft	NTV	NTV	NTV	NTV	1.03	1.13	1.01	1.24

Table 4.5: Permanent/Delay fault model: Results from experiments on two different designs - one synthesized by the RC and the other synthesized by the DC. Table shows average normalized values of $\sigma_{\mathcal{FR}}$ and $\sigma_{\mathcal{FR}}/\mu_{\mathcal{FR}}$ due to Sw and SwN transformation. NTV refers to “No Timing Violations”.

- Reduction is consistently more with IS2 than IS1.

In case of Km, multiplier instructions are very far apart - with $l_{burst} = 2500$, the number of multiplies encountered while the fault is active is only 1.1% of the total multiply instructions compared to >20% in the other applications.

For PD fault model (see Table 4.5), we observed that the number of timing violations are highly application dependent. In all timing violations, it is noted that one of the operands of the current input vector is a small positive number and the other operand is a small negative number. In Ft, usually one of the operands has a large magnitude and in Km, both the operands are always the same. Hence for both these applications no timing violations are observed (see columns 2-5 of the table). For other applications, reductions in $\sigma_{\mathcal{FR}}/\mu_{\mathcal{FR}}$ are low because as observed in Fig.2.4, conditional failure probability does not rise much untill more than 75% bits are shared, which is equivalent to very rare three or more operand sharing between pairs of input vectors. Conditional failure probability is also affected by the underlying design. We experimented with a Synopsys DC synthesized multiplier design using "carry-save-array" synthesis model and much larger reductions were observed as shown in the last four columns of the same table. With DC synthesized design, we observed 4X reduction in $\sigma_{\mathcal{FR}}/\mu_{\mathcal{FR}}$ caused by 64X reduction in $\sigma_{\mathcal{FR}}$ and 17X reduction in $\mu_{\mathcal{FR}}$, for Fr. This high impact is an extreme demonstration of the effectiveness of the proposed code transformations. In the original code of Fr, the first operand is shared more than the second operand and it was observed that first operand very strongly dictated the timing violations. Hence, compared to the transformed codes where both the operands are made to be shared equally, original code induces more extreme fault rates. Unlike in stuck-at fault model, with delay faults we do not expect SwN to improve $\sigma_{\mathcal{FR}}/\mu_{\mathcal{FR}}$ more than Sw because the concept of relaxed similarity is not applicable to delay faults due to transistor stacking effect, as mentioned in the Section 3.1.

App	$\omega_{\mathcal{FR}}^{norm,avg}$		$\omega_{\mathcal{EM}}^{norm,avg}$	
	Sw	SwN	Sw	SwN
Fr	0.88	0.88	0.93	0.93
Ac	0.92	0.78	0.96	0.88
Mm	0.85	**0.57**	0.85	**0.57**
Km	1.00	1.00	1.00	0.96
Ft	1.00	0.83	1.00	0.92

Table 4.6: Permanent/Stuck-at fault model: Average reduction in $\omega_{\mathcal{FR}}$ and $\omega_{\mathcal{EM}}$ due to Sw and SwN transformations compared to the original code. Maximum reduction of 74% is observed for Mm (shown in bold font).

4.2.2 Impact of code transformations on $\omega_{\mathcal{FR}}$ and $\omega_{\mathcal{EM}}$.

Subject to the reduction in $\mu_{\mathcal{FR}}$, reduction in $\sigma_{\mathcal{FR}}$ implies reduction in $\omega_{\mathcal{FR}}$ and, for some applications, in $\omega_{\mathcal{EM}}$ as well (see section 2.2.1). Results in Tables 4.6-4.8 show the average reductions in these two quantites for all three fault models. For the PS model, the maximum improvement observed was 74% in Mm due to combined reduction in $\sigma_{\mathcal{FR}}$ as well as $\mu_{\mathcal{FR}}$. Negligible improvement for Km and Ft is due to negligible reduction in the original similarity. For the IS model, with short fault duration (IS0) benefits are unpredictable, whereas with the larger fault durations (IS1,IS2), benefits improve from IS1 to IS2. In case of PD fault model, we experimented with two multiplier designs. With RC design, improvements in $\omega_{\mathcal{FR}}$ are smaller compared to the improvements with DC design for the same reasons as mentioned above.

4.2.3 Impact of code optimizations on efficacy of code transformations

In order to evaluate the efficacy of code transformations we compiled all our applications - both the original as well as the transformed versions, with the high-

App	$\omega_{\mathcal{FR}}^{norm,avg}$			$\omega_{\mathcal{EM}}^{norm,avg}$		
	IS0	IS1	IS2	IS0	IS1	IS2
Fr	0.98	0.88	0.86	0.99	0.92	0.90
Ac	0.78	0.80	0.79	0.85	0.90	0.89
Mm	1.01	0.98	0.93	1.00	0.98	0.93
Km	0.65	0.98	0.98	1.49	1.25	1.11
Ft	1.00	1.03	1.03	0.94	1.00	1.00

Table 4.7: Intermittent/Stuck-at fault model: Average reduction in $\omega_{\mathcal{FR}}$ and $\omega_{\mathcal{EM}}$ due to SwN transformation compared to the original code. IS0, IS1, IS2 correspond to three different burst lengths ($l_{burst} = 50, 500, 2500$).

App	RC synthesized design				DC synthesized design			
	$\omega_{\mathcal{FR}}^{norm,avg}$		$\omega_{\mathcal{EM}}^{norm,avg}$		$\omega_{\mathcal{FR}}^{norm,avg}$		$\omega_{\mathcal{EM}}^{norm,avg}$	
	Sw	SwN	Sw	SwN	Sw	SwN	Sw	SwN
Fr	0.64	0.57	0.79	0.75	0.02	0.02	0.68	0.66
Ac	1.01	0.99	1.03	1.00	0.77	0.68	0.92	0.79
Mm	0.87	0.88	0.97	0.98	0.88	0.94	0.87	0.97
Km	NTV	NTV	NTV	NTV	NTV	NTV	NTV	NTV
Ft	NTV	NTV	NTV	NTV	1.03	1.04	1.11	1.22

Table 4.8: Permanent/Delay fault model: Results from experiments on two different designs - one synthesized by the RC and the other synthesized by the DC. Average reduction in $\omega_{\mathcal{FR}}$ and $\omega_{\mathcal{EM}}$ due to Sw and SwN transformations compared to the original code. NTV refers to “No Timing Violations”.

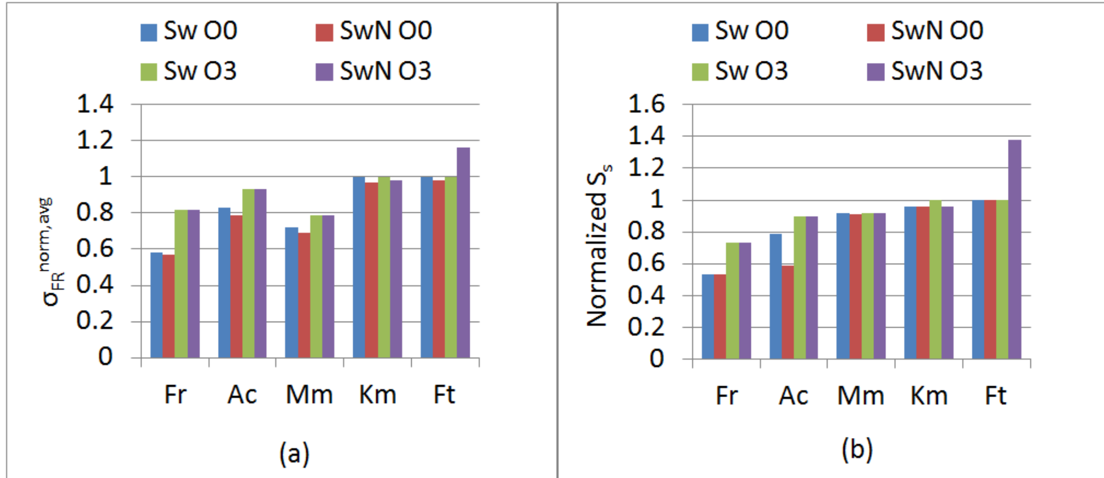

Figure 4.2: Comparing the efficacy of code transformations Sw and SwN, under optimization flags O3 and O0. (a) Reduction in standard deviation in fault rate $\sigma_{\mathcal{FR}}^{norm,avg}$, (b) Reduction in strict similarity S_s.

est optimization flag O3. We injected permanent stuck-at faults and computed $\sigma_{\mathcal{FR}}^{norm,avg}$, due to both the code transformations. Fig.4.2a compares $\sigma_{\mathcal{FR}}^{norm,avg}$ for codes compiled with optimization flag O3 and without any optimization flag (O0). Under O3, reductions in $\sigma_{\mathcal{FR}}^{norm,avg}$ are always lesser than O0. This can be explained by observing the reductions in strict similarity (S_s) as shown in the Fig.4.2b. Due to O3 optimization, strict similarity does not reduce as much as it reduces under O0. This is because compiler while unrolling the loops does not preserve the operand order as determined in the high-level code. Moreover, in order to reduce the instruction count, in case of SwN transformation, compiler reverses most of the negations which are actually redundant. Consequently, the impact of SwN is largely the same as that of Sw. For Ft, code transformations under O3 worsen $\sigma_{\mathcal{FR}}^{norm,avg}$, again in agreement with S_s. This is because Ft has the lowest similarity amongst all 5 applications, hence the profile based similarity is largely determined by the inputs to the application and the operand order determined at the compile time.

Conclusion is that, alhough 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. This is out of scope for this work.

4.2.4 Fault Injections in Adder Instructions

We briefly studied the impact of code transformations applied to the adder operations for three applications, namely, Fr, Ac and Mm with PS fault model. Other two applications could not be emulated on VarEmu as they resulted in segmentation fault in every execution.

Like Km does not have compiler visible operand sharing amongst multiply instructions, neither of the three applications have compiler visible non-constant operand sharing amongst adder instructions. Increment and comparison operations in the for-loop share constant operands which do not count towards similarity. Note that all of them are different multiply-accumulate kernels. As a result, the absolute similarity values are very small. Table 4.9 shows the strict similarity alongwith the average normalized $\sigma_{\mathcal{FR}}$ and $\sigma_{\mathcal{FR}}/\mu_{\mathcal{FR}}$. Due to low levels of similarity, firstly, profile based similarity computation is determined by the inputs and code transformations do not guarantee a consistent reduction in similarity, and secondly, the contribution of similarity towards $\sigma_{\mathcal{FR}}$ is very small. Hence, as expected, they do not correlate. Therefore, it is concluded that the "blind" transformations that are applied in absence of any operand sharing can hurt instead of benefitting.

While transformations arrest the worst-case fault rates, they increase the probability of atleast one fault activation because input vectors generated by the transformed codes are more diversified and hence, more chances of atleast one of them activating the fault. Nonetheless, results show that the transformations also improve the $\mu_{\mathcal{FR}}$. In the ongoing work we are trying to understand the dynamics

App	S_s(1e-5)			$\sigma_{\mathcal{FR}}^{norm,avg}$		$(\sigma_{\mathcal{FR}}/\mu_{\mathcal{FR}})^{norm,avg}$	
	Org	Sw	SwN	Sw	SwN	Sw	SwN
Fr	1.33	1.37(1.03)	3.46(2.60)	1.43	1.40	1.57	1.68
Ac	1.08	0.81(0.75)	2.76(2.57)	1.02	0.75	1.02	0.98
Mm	6.54	5.42(0.83)	3.63(0.55)	1.08	1.08	1.03	1.03

Table 4.9: Fault injection in adder instructions using permanent stuck-at fault model. Values inside the brackets are normalized (with respect to the original code, Org) strict similarity (S_s).

App	Ac	Ft	Fr	Km	Mm
$R_{Sw,norm}$	1.01	1.01	1.00	1.00	1.00
$R_{SwN,norm}$	1.02	1.01	1.09	0.98	1.08

Table 4.10: Runtime penalty in transformed codes. $R_{Sw,norm}$ and $R_{SwN,norm}$ are the runtimes of the transformed codes normalized with respect to the original code's runtime.

between the $\mu_{\mathcal{FR}}$ and the similarity. It demands more careful analysis because $\mu_{\mathcal{FR}}$ is a strong function of inputs and the fault location rather than being just inherent to an implementation.

Both the transformations have minimal performance overhead of $< 10\%$ as recorded in the Table 4.10. For estimating performance overhead, benchmarks were executed on the host machine (x86_64) machine instead of VarEmu because due to binary translation, runtime numbers obtained from VarEmu are not meaningful.

4.3 Chapter Summary

In this chapter we have described our hierarchical fault injection infrastructure and discussed the results of fault injection experiments with different fault models.

We specifically discuss the impact of code transformations on standard deviation in the fault rate and the worst case fault rate. Our main observations are:

- Permanent/Stuck-at fault model: Both the code transformations reduce standard deviation in the fault rate in proportion with the reduction in similarity. Improvement due to Swap-Negate code transformation is always atleast as much as due to Swap transformation.

- Intermittent/Stuck-at fault model: For small fault durations, reductions are unpredictable but as the fault duration increases, similarity better models the standard deviation in the fault rate.

- Permanent/Delay fault model: Although code transformations almost always improve the standard deviation in the fault rate and the worst case fault rate, improvements are heavily design dependent.

- Runtime penalty of both the transformations is less than 10%.

CHAPTER 5

Conclusions

5.1 Conclusions

High cost of hardware reliability mechanisms motivate a need for software based reliability mechanisms. While existing software reliability mechanisms can effectively mask or detect and recover from transient faults, they are not suitable for intermittent and permanent faults because latter are persistant and therefore, do not get usually masked. There are many symptom-based softwar monitors that detect the in-field breakdown of the hardware device. However, they rely on catastrophic symptoms and thus, silent data corruptions (SDC) escape them.

In this work, we have studied the impact of permanent and intermittent hardware failures on the SDC fault rate in software applications 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% and in one extreme case, we even observe 55X reduction, with less than 10% performance degradation.

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. For intermittent stuck-at faults correlation between similarity and standard deviation improves with the fault duration. In case of delay faults due to 1) heavy dependence on the underlying design which affects the absolute value of conditional

failure probability, and 2) highly varying average fault rate which affects the standard deviation as well, correlation is comparatively weaker.

5.2 Future Work

In the ongoing work, we are studying the impact of similarity on average fault rate. Depending upon the underlying hardware design and the location of the fault, the average fault rate can either improve, degrade or remain unchanged due to the proposed code transformations. In Fig.5.1, the fault rates due to the original as well as the transformed codes of Fr application, executing the same input, on three different faults ($F1$, $F2$ and $F3$) have been pictorially shown. Swap (Sw) and swap-negate (SwN) transformations distribute the input vectors which are originally along Y-axis, along X-axis as well as the Y-axis. In case of fault $F1$, since near X-axis not many fault activating input vectors are present, the fault rate improves by 1.67X with either transformations. In case of $F2$, fault activating input vectors are oriented along the X-axis, hence, tranformations degrade the fault rate by 1.80X. Lastly, with $F3$, fault activating input vectors are equally spread along the X and the Y-axis, hence transformations do not change the fault rate significantly. Although early observations hint that the fault rate reduces when originally it is large and the spread of the fault activating input vectors is smaller in comparison, it is not verified and needs further analysis.

In future, we would like to automatically detect the opportunities for code transformations and implement them at the compiler level. Apart from architecture independent code transformations we would explore architecture dependent code transformations. For instance, we observed that delay faults in ripple carry adder are more probable when a small positive operand is added to another small negative operand due to long propagation carry chain. We verified with some preliminary experiments that by shifting the operands to the left by small amounts

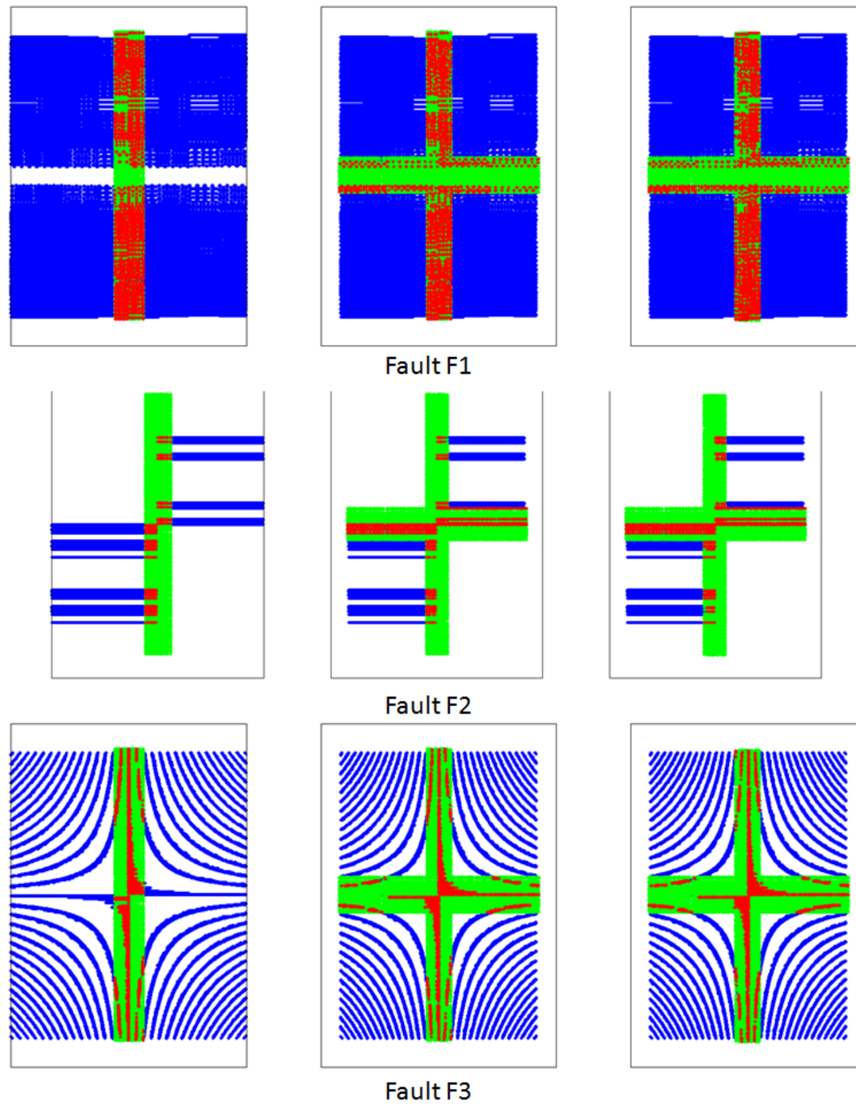

Figure 5.1: Overlap (red region) between the fault activating input vectors (blue region) and the input vectors generated (green region) by the original (left column), Sw transformed (middle column) and SwN (right column) transformed codes for three different faults - $F1$, $F2$ and $F3$.

delay faults can be exponentially reduced. Such code transformations would not only reduce the standard deviation but also the average fault rate.

References

[AFK03] Joakim Aidemark, Peter Folkesson, and Johan Karlsson. "On the probability of detecting data errors generated by permanent faults using time redundancy." In *On-Line Testing Symposium, 2003. IOLTS 2003. 9th IEEE*, pp. 68–74. IEEE, 2003.

[And79] Dorothy M Andrews. "Using executable assertions for testing and fault tolerance." In *9th Fault-Tolerance Computing Symp*, pp. 20–22, 1979.

[Ass03] JEDEC Solid State Technology Association et al. "Failure mechanisms and models for semiconductor devices." *JEDEC Publication JEP122-B*, 2003.

[Avi85] Algirdas Avizienis. "The N-version approach to fault-tolerant software." *Software Engineering, IEEE Transactions on*, (12):1491–1501, 1985.

[Bor05] Shekhar Borkar. "Designing reliable systems from unreliable components: the challenges of transistor variability and degradation." *Micro, IEEE*, **25**(6):10–16, 2005.

[BSO05] Fred A Bower, Daniel J Sorin, and Sule Ozev. "A mechanism for online diagnosis of hard faults in microprocessors." In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pp. 197–208. IEEE Computer Society, 2005.

[Cad] Cadence. "RTL Compiler.".

[CD06] Zizhong Chen and Jack Dongarra. "Algorithm-based checkpoint-free fault tolerance for parallel matrix computations on volatile resources." In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pp. 10–pp. IEEE, 2006.

[CFG05] Zizhong Chen, Graham E Fagg, Edgar Gabriel, Julien Langou, Thara Angskun, George Bosilca, and Jack Dongarra. "Fault tolerant high performance computing by a coding approach." In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 213–223. ACM, 2005.

[CLM12] Hyungmin Cho, Larkhoon Leem, and Subhasish Mitra. "ERSA: Error resilient system architecture for probabilistic applications." *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, **31**(4):546–558, 2012.

[CMA07] Kypros Constantinides, Onur Mutlu, Todd Austin, and Valeria Bertacco. "Software-based online detection of hardware defects mechanisms, architectural support, and evaluation." In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 97–108. IEEE Computer Society, 2007.

[CRA06] Jonathan Chang, George A Reis, and David I August. "Automatic instruction-level software-only recovery." In *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*, pp. 83–92. IEEE, 2006.

[CSG11] Tuck-Boon Chan, John Sartori, Puneet Gupta, and Rakesh Kumar. "On the efficacy of NBTI mitigation techniques." In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*, pp. 1–6. IEEE, 2011.

[EKD03] Dan Ernst, Nam Sung Kim, Shidhartha Das, Sanjay Pant, Rajeev Rao, Toan Pham, Conrad Ziesler, David Blaauw, Todd Austin, Krisztian Flautner, et al. "Razor: A low-power pipeline based on circuit-level timing speculation." In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, pp. 7–18. IEEE, 2003.

[FGA10] Shuguang Feng, Shantanu Gupta, Amin Ansari, and Scott Mahlke. "Shoestring: probabilistic soft error reliability on the cheap." In *ACM SIGARCH Computer Architecture News*, volume 38, pp. 385–396. ACM, 2010.

[GAM02] Pedro Gil, Jean Arlat, Henrique Madeira, Yves Crouzet, Tahar Jarboui, Karama Kanoun, Thomas Marteau, João Durães, Marco Vieira, Daniel Gil, et al. "Fault representativeness." *Deliverable ETIE2 of Dependability Benchmarking Project, IST-2000*, **25245**, 2002.

[GGL02] J Gracia, D Gil, L Lemus, and P Gil. "Studying hardware fault representativeness with VHDL models." In *Proc. of the XVII International Conference on Design of Circuits and Integrated Systems (DCIS), Santander (Spain)*, pp. 33–39, 2002.

[GRS03] Olga Goloubeva, Maurizio Rebaudengo, M Sonza Reorda, and Massimo Violante. "Soft-error detection using control flow assertions." In *Defect and Fault Tolerance in VLSI Systems, 2003. Proceedings. 18th IEEE International Symposium on*, pp. 581–588. IEEE, 2003.

[GSB08] J Gracia, L Saiz, JC Baraza, D Gil, and P Gil. "Analysis of the influence of intermittent faults in a microcontroller." In *Design and Diagnostics of Electronic Circuits and Systems, 2008. DDECS 2008. 11th IEEE Workshop on*, pp. 1–6. IEEE, 2008.

[HA84] Kuang-Hua Huang and Jacob A. Abraham. “Algorithm-based fault tolerance for matrix operations.” *Computers, IEEE Transactions on*, **100**(6):518–528, 1984.

[HAN12a] Siva Kumar Sastry Hari, Sarita V Adve, and Helia Naeimi. “Low-cost program-level detectors for reducing silent data corruptions.” In *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*, pp. 1–12. IEEE, 2012.

[HAN12b] Siva Kumar Sastry Hari, Sarita V Adve, Helia Naeimi, and Pradeep Ramachandran. “Relyzer: exploiting application-level fault equivalence to analyze application resiliency to transient faults.” In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, pp. 123–134. ACM, 2012.

[HDP06] V Huard, M Denais, and C Parthasarathy. “NBTI degradation: From physical mechanisms to modelling.” *Microelectronics Reliability*, **46**(1):1–23, 2006.

[HK89] Paul S Ho and Thomas Kwok. “Electromigration in metals.” *Reports on Progress in Physics*, **52**(3):301, 1989.

[HLD05] Jie S Hu, Feihui Li, Vijay Degalahal, Mahmut Kandemir, Narayanan Vijaykrishnan, and Mary J Irwin. “Compiler-directed instruction duplication for soft error detection.” In *Design, Automation and Test in Europe, 2005. Proceedings*, pp. 1056–1057. IEEE, 2005.

[HRR99] C-K Hu, R Rosenberg, HS Rathore, DB Nguyen, and B Agarwala. “Scaling effect on electromigration in on-chip Cu wiring.” In *Interconnect Technology, 1999. IEEE International Conference*, pp. 267–269. IEEE, 1999.

[KIR99] Zbigniew Kalbarczyk, Ravishankar K. Iyer, Gregory L. Ries, Jaqdish U. Patel, Myeong S. Lee, and Yuxiao Xiao. “Hierarchical simulation approach to accurate fault modeling for system dependability evaluation.” *Software Engineering, IEEE Transactions on*, **25**(5):619–632, 1999.

[LRK09] Man-Lap Li, Pradeep Ramachandran, Ulya R Karpuzcu, Siva Hari, and Sarita V Adve. “Accurate microarchitecture-level fault modeling for studying hardware faults.” In *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, pp. 105–116. IEEE, 2009.

[LRS08a] Man-Lap Li, Pradeep Ramachandran, Swarup K Sahoo, Sarita V Adve, Vikram S Adve, and Yuanyuan Zhou. "SWAT: An error resilient system." In *4th Workshop on Silicon Errors in Logic-System Effects*, 2008.

[LRS08b] Man-Lap Li, Pradeep Ramachandran, Swarup Kumar Sahoo, Sarita V Adve, Vikram S Adve, and Yuanyuan Zhou. "Understanding the propagation of hard errors to software and implications for resilient system design." *ACM Sigplan Notices*, **43**(3):265–276, 2008.

[MLN02] Shahrzad Mirkhani, Meisam Lavasani, and Zainalabedin Navabi. "Hierarchical fault simulation using behavioral and gate level hardware models." In *Test Symposium, 2002.(ATS'02). Proceedings of the 11th Asian*, pp. 374–379. IEEE, 2002.

[MPR00] S Mahapatra, Chetan D Parikh, V Ramgopal Rao, Chand R Viswanathan, and Juzer Vasi. "Device scaling effects on hot-carrier induced interface and oxide-trapped charge distributions in MOSFETs." *IEEE Transactions on Electron Devices*, **47**(4):789–796, 2000.

[NKS12] Sani R Nassif, Veit B Kleeberger, and Ulf Schlichtmann. "Goldilocks failures: Not too soft, not too hard." In *Reliability Physics Symposium (IRPS), 2012 IEEE International*, pp. 2F–1. IEEE, 2012.

[OM01] Nahmsuk Oh and Edward J McCluskey. "Procedure call duplication: minimization of energy consumption with constrained error detection latency." In *Defect and Fault Tolerance in VLSI Systems, 2001. Proceedings. 2001 IEEE International Symposium on*, pp. 182–187. IEEE, 2001.

[OSM02] Nahmsuk Oh, Philip P Shirvani, and Edward J McCluskey. "Error detection by duplicated instructions in super-scalar processors." *Reliability, IEEE Transactions on*, **51**(1):63–75, 2002.

[PBM00] Marius Pirvu, Laxmi Bhuyan, and Rabi Mahapatra. "Hierarchical simulation of a multiprocessor architecture." In *Computer Design, 2000. Proceedings. 2000 International Conference on*, pp. 585–588. IEEE, 2000.

[PKI05] Karthik Pattabiraman, Zbigniew Kalbarczyk, and Ravishankar K Iyer. "Application-based metrics for strategic placement of detectors." In *Dependable Computing, 2005. Proceedings. 11th Pacific Rim International Symposium on*, pp. 8–pp. IEEE, 2005.

[Ran75] Brian Randell. "System structure for software fault tolerance." *Software Engineering, IEEE Transactions on*, (2):220–232, 1975.

[RCV05] George A Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I August. “SWIFT: Software implemented fault tolerance.” In *Proceedings of the international symposium on Code generation and optimization*, pp. 243–254. IEEE Computer Society, 2005.

[rev] Omitted for blind review.

[RPG10] Layali Rashid, Karthik Pattabiraman, and Sathish Gopalakrishnan. “Towards understanding the effects of intermittent hardware faults on programs.” In *Dependable Systems and Networks Workshops (DSN-W), 2010 International Conference on*, pp. 101–106. IEEE, 2010.

[RSK11] Semeen Rehman, Muhammad Shafique, Florian Kriebel, and Jörg Henkel. “Reliable software for unreliable hardware: embedded code generation aiming at reliability.” In *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pp. 237–246. ACM, 2011.

[RSK12] S Rehman, M Shafique, F Kriebel, and J Henkel. “RAISE: Reliability-Aware Instruction SchEduling for Unreliable Hardware.” In *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*, pp. 671–676. IEEE, 2012.

[SAB04] Jayanth Srinivasan, Sarita V Adve, Pradip Bose, and Jude A Rivers. “The impact of technology scaling on lifetime reliability.” In *Dependable Systems and Networks, 2004 International Conference on*, pp. 177–186. IEEE, 2004.

[Sah] Goutam Kumar Saha. “Software based fault tolerance: a survey.”.

[Sah06] Goutam Kumar Saha. “Application semantic driven assertions toward fault tolerant computing.” *Ubiquity*, **2006**(June):1, 2006.

[SGH07] Jared C Smolens, Brian T Gold, James C Hoe, Babak Falsafi, and Ken Mai. “Detecting emerging wearout faults.” In *Proc. of Workshop on SELSE*, 2007.

[SK09] Vilas Sridharan and David R Kaeli. “Eliminating microarchitectural dependency from architectural vulnerability.” In *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, pp. 117–128. IEEE, 2009.

[SLR08] Swarup Kumar Sahoo, Man-Lap Li, Pradeep Ramachandran, Sarita V Adve, Vikram S Adve, and Yuanyuan Zhou. “Using likely program invariants to detect hardware errors.” In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pp. 70–79. IEEE, 2008.

[SSK11] John Sartori, Joseph Sloan, and Rakesh Kumar. "Stochastic computing: Embracing errors in architecture and design of processors and applications." In *Compilers, Architectures and Synthesis for Embedded Systems (CASES), 2011 Proceedings of the 14th International Conference on*, pp. 135–144. IEEE, 2011.

[Sta02] James H Stathis. "Reliability limits for the gate insulator in CMOS technology." *IBM Journal of Research and Development*, **46**(2.3):265–286, 2002.

[Syn] Synopsys. "Design Compiler.".

[TR84] Asser N Tantawi and Manfred Ruschitzka. "Performance analysis of checkpointing strategies." *ACM Transactions on Computer Systems (TOCS)*, **2**(2):123–144, 1984.

[WHV95] Yi-Min Wang, Yennun Huang, Kiem-Phong Vo, Pe-Yu Chung, and Chandra Kintala. "Checkpointing and its applications." In *Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on*, pp. 22–31. IEEE, 1995.

[WP06] Nicholas J Wang and Sanjay J Patel. "ReStore: Symptom-based soft error detection in microprocessors." *Dependable and Secure Computing, IEEE Transactions on*, **3**(3):188–201, 2006.

[WRP11] Jiesheng Wei, Layali Rashid, Karthik Pattabiraman, and Sathish Gopalakrishnan. "Comparing the effects of intermittent and transient hardware faults on programs." In *Dependable Systems and Networks Workshops (DSN-W), 2011 IEEE/IFIP 41st International Conference on*, pp. 53–58. IEEE, 2011.

[ZKE12] Samy Zaynoun, Muhammad S Khairy, Ahmed M Eltawil, Fadi J Kurdahi, and Amin Khajeh. "Fast error aware model for arithmetic and logic circuits." In *Computer Design (ICCD), 2012 IEEE 30th International Conference on*, pp. 322–328. IEEE, 2012.