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

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

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

I. Introduction

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

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

Funding for this research was provided by the NSF under grant number CCF-1029030.

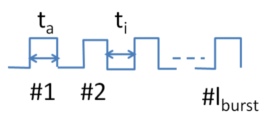


Fig. 1: Intermittent fault model parameters, adapted from [8]

```
// Sample Code

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

Fig. 2: A sample code

analyze the impact of intermittent faults on programs in terms of crashes and hangs, whereas we focus on SDC.

Our major contributions are

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

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

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

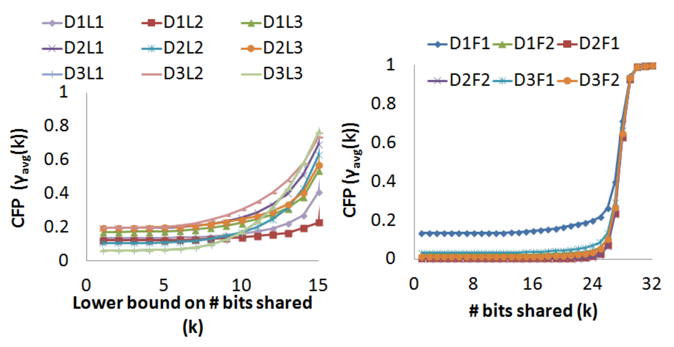
II. Fault Models

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

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

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

Transient fault activation which is usually modeled as a single event upset is time dependent. At the gate-level, a fault is said to be activated if any one of the input or the output bits of the faulty gate is flipped. Whereas, since permanent faults perpetually exist, their activatation is solely dependent on the system state, independent of



(a) Stuck-At Faults (b) Delay Faults

Fig. 3: Average conditional failure probability (CFP, $\gamma_{avg}(k)$) as a function of the number of bits shared (k) between input vectors for 8-bit multiplier designs. (a) Stuck-at fault model: 3 different designs (D) and 3 different randomly chosen nets as fault locations (L). (b) Delay fault model: 3 different designs (D) and 2 different frequency overscaling factors (F) - 10% (F1) and 5% (F2). Note that for delay faults we compute bit sharing between two pairs of consequtive input vectors. Hence, they can share upto 32 bits.

the time. Time independence in case of intermittent fault activation is partial because faults do not perpetually exist.

Under stuck-at fault model, state dependence can be interpreted as dependence on the current input vector. For instance, consider a 2-input AND gate - $\{a, b\}$ are the inputs and z is the output. If z is stuck-at 0, fault can be activated only when $a = b = 1$. In other words, this stuck-at fault can be characterized by a boolean expression, ab, such that the fault is activated if and only if this expression is satisfied. Under the delay fault model, however, the fault activation depends on the current as well as the previous input vector [31]. Timing violations are contingent upon the sensitization of critical paths which in turn depends on two consequtive input vectors, assuming the faulty unit is isolated and does not receive off path inputs. Exploiting this input vector dependence, we make the following observation:

The conditional failure probability, defined for two vectors[1] v_i *and* v_j*, increases as they share more bits. The conditional failure probability (CFP) is the probability that* v_j *activates a fault given that* v_i *activates the same fault or vice-versa.*

This should not be surprising because the conditional probability of satisfying the boolean expression increases with the increasing number of shared bits. This observation has been empirically confirmed through verilog simulations on three synthesized gate-level 8-bit multiplier designs. One of them is a general design synthesized using Cadence RTL Compiler [5] (D1) and other two are synthesized using Synopsys Design Compiler [6], 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.

Fig.3a and 3b, plot CFP as a function of (lower bound on) the number of bits shared (k) between input vectors. For instance, $k = 8$ means atleast 8 bits are shared. CFP is computed as the ratio of 1) the average number of **fault activating input vectors** that share atleast k bits with a given fault activating input vector, and 2) the average number of **input vectors** that share atleast k bits with any given fault activating input vector. CFP rises exponentially with k for both the fault models, but with delay faults it is almost flat until $k = 22$

[1]In case of delay faults, instead of "input vectors" we should always consider "a pair of consequtive input vectors"

```
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;
}
```

(a)

```
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] = t2 * (-f[i]);
}
```

(b)

Fig. 4: Code Transformations for the original code of Fig.2. (a) Transformation *Swap* (Sw): Half the operands are swapped (b) Transformation *Swap-Negate* (SwN): Half the operands are swapped and multiplied by -1.

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

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

III. Similarity Metric and Code Transformations

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

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

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

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

Using the procedure outlined in Eqn.(1), similarity which is inherent to a code, can be statically computed. For instance, consider the kernel shown in Fig.2. N input vectors share the first operand with the value e. Thus, there are $\binom{N(N-1)}{2}$ pairs sharing an operand. Therefore, for this original code (Org), $S_{Org} = \frac{N-1}{2N}$. 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 fault-free profiled executions needs to be computed. Since one of the benchmark kernels has zero compiler visible operand sharing amongst multiply instructions, profiling has been used to compute similarity.

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

App	Fr	Ac	Mm	Km	Ft
S_{Org}	14.9	5.50	4.70	0.40	0.04
S_{Sw}	7.95(0.53)	4.36(0.79)	4.31(0.91)	0.38(0.96)	0.04(1.00)
S_{SwN}	7.94(0.53)	3.25(0.59)	4.30(0.91)	0.37(0.96)	0.04(1.00)

TABLE I: Similarity values (in the order of 1e-3) for the original (Org), swapped (Sw) and swap-negated (SwN) codes of all the applications are shown. Normalized (with respect to Org) similarity values are shown in the brackets. Number of faulty input vectors, $N \sim 42000$

vectors sharing a particular operand, although there are two sets of them. Therefore, when statically computed, similarity would be,

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

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

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

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

IV. Experimental Setup and Results

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

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

PS fault model: There are two important observations:

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

App	S		$\sigma_{\mathcal{FR}}$		$\omega_{\mathcal{FR}}$	
	Sw	SwN	Sw	SwN	Sw	SwN
Fr	0.53	0.53	0.58	0.57	0.88	0.88
Ac	0.79	0.59	0.83	0.79	0.92	0.78
Mm	0.92	0.91	0.72	0.69	0.85	0.57
Km	0.96	0.96	1.00	0.97	1.00	1.00
Ft	1.00	1.00	1.00	0.98	1.00	0.92

TABLE II: Permanent/Stuck-at fault model: Correlating reduction in similarity (S) and reduction in $\sigma_{\mathcal{FR}}$ due to Sw and SwN transformations. Also shown are the worst-case fault rates $\omega_{\mathcal{FR}}$ of transformed codes. All figures are normalized with respect to the original code.

App	$\sigma_{\mathcal{FR}}$			$\omega_{\mathcal{FR}}$		
	IS0	IS1	IS2	IS0	IS1	IS2
Fr	1.05	0.75	0.61	0.98	0.88	0.86
Ac	0.72	0.81	0.77	0.78	0.80	0.79
Mm	1.15	1.28	0.94	1.01	0.98	0.93
Km	0.80	1.10	1.10	0.65	0.98	0.98
Ft	0.93	1.31	1.07	1.00	1.03	1.03

TABLE III: Intermittent/Stuck-at fault model: Table shows normalized values of $\sigma_{\mathcal{FR}}$ and $\omega_{\mathcal{FR}}$ 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.

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

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

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

Although the results show that the transformations improve $\mu_{\mathcal{FR}}$, we need to understand the dynamics between $\mu_{\mathcal{FR}}$ and the similarity.

App	$\sigma_{\mathcal{FR}}$		$\omega_{\mathcal{FR}}$			$\omega_{\mathcal{FR}}$ (DC synth)	
	Sw	SwN	Sw	SwN		Sw	SwN
Fr	0.62	0.53	0.64	0.57		0.02	0.02
Ac	1.01	0.98	1.01	0.99		0.77	0.68
Mm	0.66	0.67	0.87	0.88		0.88	0.94

TABLE IV: Permanent/Delay fault model: Table shows normalized values of $\sigma_{\mathcal{FR}}$ and $\omega_{\mathcal{FR}}$ due to Sw and SwN transformation. For Km and Ft, no faults are observed. Last two columns show results from a DC synthesized design.

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 V: 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.

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. Though we applied our code transformations by modifying the high-level code, compiler can likely do a better job by applying transformations automatically since it can preserve them through the compiler optimizations.

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

V. Conclusions and Future Work

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

VI. Acknowledgement

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

References

[1] Joakim Aidemark et al. On the probability of detecting data errors generated by permanent faults using time redundancy. In *IOLTS 2003*, pages 68–74. IEEE.

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

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

[4] Hyungmin Cho et al. Ersa: Error resilient system architecture for probabilistic applications. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 31(4):546–558, 2012.

[5] Cadence RTL Compiler. http://www.cadence.com/.

[6] Synopsys Design Compiler. http://www.synopsys.com/.

[7] Pedro Gil et al. Fault representativeness. *Deliverable ETIE2 of Dependability Benchmarking Project, IST-2000*, 25245, 2002.

[8] J Gracia et al. 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*, pages 1–6. IEEE, 2008.

[9] Siva Kumar Sastry Hari et al. Low-cost program-level detectors for reducing silent data corruptions. In *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*, pages 1–12. IEEE, 2012.

[10] Siva Kumar Sastry Hari et al. Relyzer: exploiting application-level fault equivalence to analyze application resiliency to transient faults. In *International conference on Architectural Support for Programming Languages and Operating Systems*, pages 123–134. ACM, 2012.

[11] Kuang-Hua Huang et al. Algorithm-based fault tolerance for matrix operations. *Computers, IEEE Transactions on*, 100(6):518–528, 1984.

[12] Man-Lap Li et al. Swat: An error resilient system. In *4th Workshop on Silicon Errors in Logic-System Effects*, 2008.

[13] Man-Lap Li et al. Accurate microarchitecture-level fault modeling for studying hardware faults. In *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, pages 105–116. IEEE, 2009.

[14] Sani R Nassif et al. Goldilocks failures: Not too soft, not too hard. In *Reliability Physics Symposium (IRPS), 2012 IEEE International*, pages 2F–1. IEEE, 2012.

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

[16] Layali Rashid et al. Towards understanding the effects of intermittent hardware faults on programs. In *Dependable Systems and Networks Workshops (DSN-W), 2010 International Conference on*, pages 101–106. IEEE, 2010.

[17] S Rehman et al. Raise: Reliability-aware instruction scheduling for unreliable hardware. In *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*, pages 671–676. IEEE, 2012.

[18] Semeen Rehman et al. 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*, pages 237–246. ACM, 2011.

[19] George A Reis et al. Swift: Software implemented fault tolerance. In *Proceedings of the international symposium on Code generation and optimization*, pages 243–254. IEEE Computer Society, 2005.

[20] Goutam Kumar Saha. Software based fault tolerance: a survey.

[21] Swarup Kumar Sahoo et al. Using likely program invariants to detect hardware errors. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 70–79. IEEE, 2008.

[22] John Sartori et al. 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*, pages 135–144. IEEE, 2011.

[23] Ankur Sharma. Understanding software application behaviour in presence of permanent and intermittent hardware faults. In *Tech. Rep. http://nanocad.ee.ucla.edu/pub/Main/Publications/MSTH3_paper.pdf.* UCLA, 2013.

[24] Joseph Sloan et al. A numerical optimization-based methodology for application robustification: Transforming applications for error tolerance. In *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*, pages 161–170. IEEE, 2010.

[25] Jared C Smolens et al. Detecting emerging wearout faults. In *Proc. of Workshop on SELSE*, 2007.

[26] Jayanth Srinivasan et al. The impact of technology scaling on lifetime reliability. In *Dependable Systems and Networks, 2004 International Conference on*, pages 177–186. IEEE, 2004.

[27] Asser N Tantawi et al. Performance analysis of checkpointing strategies. *ACM Transactions on Computer Systems (TOCS)*, 2(2):123–144, 1984.

[28] Yi-Min Wang et al. Checkpointing and its applications. In *Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on*, pages 22–31. IEEE, 1995.

[29] Lucas F Wanner et al. Varemu: An emulation testbed for variability-aware software. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2013.

[30] Jiesheng Wei et al. 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*, pages 53–58. IEEE, 2011.

[31] Samy Zaynoun et al. Fast error aware model for arithmetic and logic circuits. In *Computer Design (ICCD), 2012 IEEE 30th International Conference on*, pages 322–328. IEEE, 2012.