Towards Understanding the Effects of Intermittent Hardware Faults on Programs

Layali Rashid, Karthik Pattabiraman and Sathish Gopalakrishnan
The University of British Columbia, Canada
{lrashid, karthikp, sathish}@ece.ubc.ca

Abstract

Intermittent hardware faults are bursts of errors that last from a few CPU cycles to a few seconds. They are caused by process variations, circuit wear-out, and temperature, clock or voltage fluctuations. Recent studies show that intermittent fault rates are increasing due to technology scaling and are likely to be a significant concern in future systems. We study the propagation of intermittent faults to programs; in particular, we are interested in the crash behaviour of programs. We use a model of a program that represents the data dependencies in a fault-free trace of the program and we analyze this model to glean some information about the length of intermittent faults and their effect on the program under specific fault and crash models. The results of our study can aid fault detection, diagnosis and recovery techniques.

1. Introduction

Intermittent hardware faults are bursts of errors that occur at the same location (micro-architectural component) and last from a few cycles to a few seconds, depending on their causes [1]. Although recent advances in semiconductor technology have led to faster devices, there has been a decline in the reliability of semiconductor devices [2]. Studies have shown that process variation, supply voltage and temperature (PVT) fluctuations increase the rates of hardware faults and these rates will only worsen over time [3]. Further, in-progress wear-out and residual manufacturing errors are likely to result in more intermittent faults [1, 4], which may eventually become permanent faults [1]. Intermittent faults, therefore, are likely to be a significant concern in future processors.

An intermittent fault, unlike a permanent fault, does not persist and is hard to diagnose post facto using (hardware/software) tests because the conditions that caused the fault are hard to regenerate [5]. Further, it has been suggested that intermittent faults have the potential to impact program execution to a greater extent when compared with transient faults [1].

We study the propagation of intermittent faults in programs. This work is important in understanding error propagation and, hence, in implementing fault detection, diagnosis and recovery techniques. While intermittent faults may manifest in programs in many ways, we focus on fault bursts that affect multiple consecutive instructions in an execution of a program.

To the best of our knowledge, there has been no study on intermittent fault propagation at the program-level. Prior work has investigated the behavior of intermittent faults using a VHDL representation of a commercial microcontroller [6], using micro-architectural models to understand the effects of permanent and transient faults on software [7-8], or by conducting massive fault-injection campaigns to study the behaviour of transient faults [9]. However, none of these studies examined the impact of intermittent faults on programs (to the best of our knowledge).

In this work, we perform a set of empirical measurements to characterize the behaviour of intermittent faults. An intermittent fault, for the purposes of our study, can be specified by the program instruction that is being executed when the fault starts and by the number of instructions (the length) over which the fault persists. In this preliminary study, we examine two benchmark programs in detail to understand the following research questions:

1. Do all intermittent faults lead to a program crash?
2. How many instructions are executed from the start of the intermittent fault and before the program crashes? How does this metric vary with respect to the length of the intermittent fault?
3. How many erroneous values are generated after the start of the intermittent fault and before the program crashes (due to the fault)?

We find that most intermittent faults (~95% for both programs) cause the program to crash within 10 dynamic instructions from the beginning of the fault. Further, our study shows that intermittent faults that persist for less than 5 instructions are more likely to propagate through the program. Faults of longer duration cause programs to crash quickly, i.e., longer faults lead to crashes when the fault is active or soon

after the end of the fault. In both cases, only a limited number of data values in the program are corrupted before the crash (16 or less in our experiments).

We use a high-level model of the program's fault-free run to learn about dynamic data dependencies and to understand the propagation of intermittent faults via corrupted data in programs (details in Section 2). Our model is more efficient than a fault-injection campaign. We also evaluate the accuracy of crash locations found by the model by injecting intermittent faults and simulating program execution using a modified version of the SimpleScalar simulator [10]. We find that about 88% of the crash locations are within 10 dynamic instructions of the crash locations predicted by our model.

2. Method

In this section: (1) we describe our fault model, and build a crash model to reason about where programs crash due to intermittent faults, (2) we define various metrics to characterize the propagation, and (3) wedescribe an example using a Dynamic Dependency Graph (DDG) [11] to understand the propagation of intermittent faults in programs.

2.1. Fault Model

We consider the following faults in this study:
(1) Instruction decoding errors that result in a different instruction being executed. However, we assume that the flow control of the program (order of instructions during program execution) is preserved.
(2) ALU errors that affect the computational operations and produce erroneous results. This includes errors in source registers, destination registers and the ALU's combinational circuitry.
(3) Load/store unit errors that result in loading/storing data from/to incorrect memory locations. We do not consider errors in loaded/stored values.
We assume that the device's memory hierarchy is reliable (register file, caches and main memory). This is reasonable as such errors can be detected by parity or ECC (if present). Further, we assume that intermittent faults span over multiple consecutive dynamic instructions. Finally, we assume that the processor's control logic is error-free.

2.2. Crash Model

The crash model describes the situations that will lead to a program crash. The model approximates actual events that may result in a program crash. It is important to build an accurate model of when the program crashes because error propagation stops at the instance of program crash. We focus on crashes because, when encountering an error, programs do not necessarily crash immediately but often continue executing, leading to error propagation [9]. Hence, crashes are not always benign or fail-stop.

In this subsection we describe our crash model and then we validate the model empirically (Section 4). We assume that a program crashes when a fault propagates to (1) memory-address in a load or store instruction, (2) destination address of a branch instruction (3) destination and return addresses in function calls. Previous studies (e.g., [12]) have shown that the execution of instructions that utilize erroneous pointer values lead to program crashes at the same instruction with high probability.

2.3. Definitions and Assumptions

This section defines the terms used in this paper.
Node: A value produced by a dynamic instruction during the program execution. A node can be a data value or a memory address. Any node can be read multiple times but it is written only once.
Dynamic Dependency Graph (DDG): A directed acyclic graph that models the dynamic dependencies between nodes generated during the program execution. It consists of the dynamic instructions of the program taken from a correct program control flow as its edges, and these edges connect its nodes [11, 13].
Transient Propagation Set or *TPS(i)* denotes the set of nodes that are written using erroneous values if a transient fault occurs at i, where i is a node generated by a dynamic instruction from a program trace. We assume an error does not propagate beyond a crash point (as predicted by the crash model).
CrashNode(i) denotes the node at which the program crashes if a transient fault occurs at the node i.
Intermittent Propagation Set or *IPS(i_s, i_e)* denotes the set of nodes that are written using erroneous values if an intermittent fault starts at i_s and ends at i_e, where i_s and i_e are nodes generated by dynamic instructions from a program trace such that $i_s < i_e$. We assume an error does not propagate beyond a crash point.
CrashNode(i_s, i_e) denotes the node at which the program crashes if an intermittent fault occurs between node i_s and node i_e.
CrashDistance(i_s, i_e) denotes the number of nodes that are generated from the time an intermittent fault occurs between node i_s and node i_e until the program crashes at node *CrashNode(i_s, i_e)*.

2.4. Computing the Intermittent Fault Propagation Set

In this section, we will demonstrate how to compute the TPS and IPS using an example code fragment (Table 1). Note that, for this example only, we assume a loop-free program and hence there is a one-to-one

mapping between nodes and instructions. We assume that the program's inputs are known a priori.

Table 1: Example code fragment.

Code Fragment	Explanation	Node
mov R1, #5	R1 ←5	1
mov R2, #6	R2 ← 6	2
mov R3, #7	R3 ← 7	3
ld R4, R1, Array_Addr	R4 ← [R1+Array_Addr]	4
ld R5, R2, Array_Addr	R5 ← [R2+Array_Addr]	5
ld R6, R3, Array_Addr	R6 ← [R3+Array_Addr]	6
mult R7, R5, R4	R7 ← R5xR4	7
add R8, R7, R6	R8 ← R7xR6	8

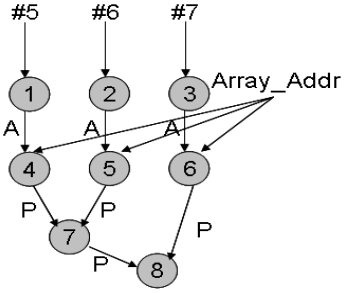

Figure 1: DDG corresponding to code fragment in Table 1.

In the example code fragment (Table 1), elements at indices 5, 6 and 7 of an array (that starts at address Array_Addr) are loaded into registers R4, R5 and R6, respectively. Then, the first two registers are multiplied and the result is stored in R7. Finally, registers R7 and R6 are added and the result is stored in R8.

A DDG (Figure 1) is constructed using established techniques [11] from a trace file of a fault-free execution of the code fragment. In the figure, nodes are drawn as circles, and the number inside the circle is the node number. The edges represent the instructions' operands and are labeled with the operand's type, where A is an address operand and P is a regular operand.

We use the example to illustrate intermittent fault propagation. Assume that an intermittent fault affects the first two instructions of the code and we need to compute *IPS(1, 2)*. Since we use *TPS(i), i = {1, 2}* to compute *IPS(1, 2)*, we first explain how to compute *TPS(i).* Recall from the previous subsection that *TPS(1)* is the transient propagation set if an error occurs at node 1. A *TPS(i)* includes at least the node at which the error occurs, in this case node 1. Node 1 is used to compute node 4, hence *TPS(1) = {1, 4}.* However, node 1 is used as an address in the fourth instruction (ld R4, R1, Array_Addr), hence the program crashes at this instruction (as per the crash model). Therefore, *CrashNode(1) = 4* and *TPS(1) = {1, 4}*. Similarly, we can find *TPS(2)* which computes *TPS* when a transient fault is injected into node 2. Since node 2 is used to generate node 5, *TPS(2) = {2, 5}*. Node 2 is used as an address in a load instruction and hence the program (potentially) crashes at this instruction. Therefore *CrashNode(2) = 5*.

To compute *IPS(1, 2)*, we find the *CrashNode(1, 2)*, which is the minimum crash node of the crash nodes for all nodes between *1* and *2* inclusive, since the program will crash and not proceed beyond this node. In this case, *min(CrashNode(1), CrashNode(2)) = CrashNode(1) = 4.* Then, we find the union of *TPS(1)* and *TPS(2)* such that all the resultant nodes occur at or before *CrashNode(1)*. Consequently, *IPS(1, 2) = {1, 2, 4}*. Note that node 5 is generated after *CrashNode(1, 2)*, and hence it is not included in *IPS(1, 2*) even though it was included in *TPS(2)*. To compute *CrashDistance(1, 2),* we count all nodes generated from the start of the intermittent fault until the program crashes, i.e., *Cardinality(CrashDistance(1, 2)) = Cardinality({1, 2, 3, 4}) = 4* nodes.

3. Experimental Setup

In this section, we describe the experimental setup to study the impact of intermittent faults. We choose two test programs: matrix multiply and insertions sort. Matrix multiply reads two matrices from memory, find their product and stores it back to memory, while Insertion Sort arranges an array of integers in ascending order. Both programs consist of approximately 11,000 assembly instructions including loops, memory instructions and I/O instructions. The total number of nodes in matrix multiply is 19029 nodes, while the total number of nodes in Insertion Sort is 16195 nodes.

Evaluation of the model's accuracy: To assess the accuracy of the crash locations found by our model (Section 2), we inject all faults expected to cause crashes by our model, one at a time, into runs on a modified version of the SimpleScalar simulator [10]. For each one of these intermittent faults, SimpleScalar executes the application and injects a single fault (a random bit-flip) into each node of the corresponding list of nodes. If the program crashes, SimpleScalar generates a crash dump file that contains the node at which the program crashed. Based on this crash dump, the difference (in number of nodes) between the crash location estimated by the model and the actual crash location is computed.

Computation of intermittent fault propagation: We modified SimpleScalar [10] to capture a trace of a fault-free program execution. For each instruction executed by the program, SimpleScalar records the instruction's type, the node(s) generated by the instruction and a list of immediate node successors to a trace file. It also records the order in which the

dynamic instructions are executed by the program (i.e., the program's control flow).

To construct the DDG model (Section 2), we need an execution trace of the program (obtained from sample run using SimpleScalar) and a list of start and end nodes of potential intermittent faults. For each fault, our approach calculates the intermittent propagation set (IPS) and the expected crash distance (if the fault results in a crash). If a crash node is encountered at the end of an intermittent fault then longer intermittent faults lengths with the same start node are not considered in the DDG model. In other words, the intermittent fault's period cannot include a crash node unless it is the last one. No fault-injections are performed in this part of the experiment.

4. Results

First we evaluate the accuracy of the crash distances predicted by the DDG model and then empirically answer the central research questions (Section 1).

We measure the total number of intermittent faults considered by the DDG model, the total number of intermittent faults expected to cause crashes and the intermittent faults that actually cause crashes in SimpleScalar (Table 2). It takes the DDG model about 3 seconds to obtain the results reported in this section. However, the SimpleScalar experiments take about 40 minutes to inject the same set of faults in each program. This result demonstrates the scalability of the DDG model over the SimpleScalar-based fault injections for the studied applications.

Question 1. Do all intermittent faults lead to a program crash?

Our DDG model found that 99% of the faults considered are expected to cause crashes in both Insertion Sort and matrix multiply (Table 2). The other 1% of the faults either resulted in Silent-Data-Corruptions (SDC) or benign outputs.

Further, we find that 43% cause crashes in Insertion Sort and 49% cause crashes in matrix multiply when the list of intermittent fault expected to cause crashes is actually injected by SimpleScalar (Table 2). This difference is because the crash model in Section 2.2 is conservative and hence over-estimates the number of crashes that occur in reality (i.e., in SimpleScalar).

Table 2: The absolute numbers of intermittent faults and their consequences in DDG and SimpleScalar

Metric(# of intermittent faults)	Insertion Sort	Matrix Multiply
Total in DDG model	15266	14543
Cause crashes in DDG model	15229	14417
Cause crashes in SimpleScalar	6549	7043

The results in the rest of this section pertain to the intermittent faults that are expected to cause crashes by the DDG model and actually do so in SimpleScalar.

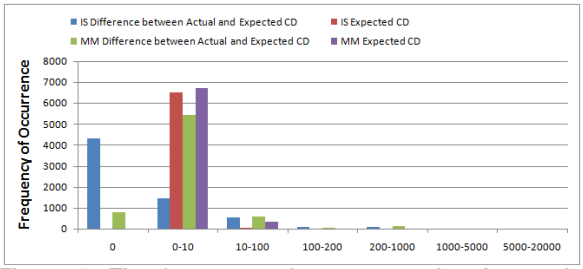

Figure 2: The frequency of occurrence for the crash distance (CD) as computed in DDG model and the difference between this crash distance and the corresponding actual crash distance. IS is Insertion Sort. MM is matrix multiply.

To evaluate the accuracy of crash distances predicted by our model, we compare the crash distances from the injected faults in SimpleScalar to those predicted by the DDG model (Figure 2). We find that the majority of the predicted crash distances (88% for Insertion Sort and 89% for matrix multiply) fall within 10 nodes from the actual ones. Moreover, 97% fall within 100 nodes of the predicted crash distance for both programs. This shows that the DDG model is accurate in predicting the crash distances.

Question 2. How many instructions are executed from the start of the intermittent fault and before the program crashes? How does this metric vary with respect to the length of the intermittent fault?

To answer this question, we measure the total number of correct and erroneous nodes generated after the start of the intermittent fault and before the program crash (i.e. crash distance, Section 2.3). We then plot the crash distances against the frequency of occurrence (Figure 2) and the intermittent fault length (Figure 3 and 4). We find that 95% and 99% of the crash distances fall within 10 nodes of the start of the intermittent fault for matrix multiply and Insertion Sort, respectively. A previous study by Gu et al. found similar behavior but for transient errors,i.e., they show that most transient errors injected into Linux-kernel cause a crash within 10 cycles [9]. The remaining 5% and 1% of the intermittent faults cause the program to crash within 100 nodes of the intermittent fault's start.

In summary, most intermittent faults (~95% for both programs) cause program to crash within 10 nodes of the fault's start. Thus, large crash distances are infrequent in the programs studied.

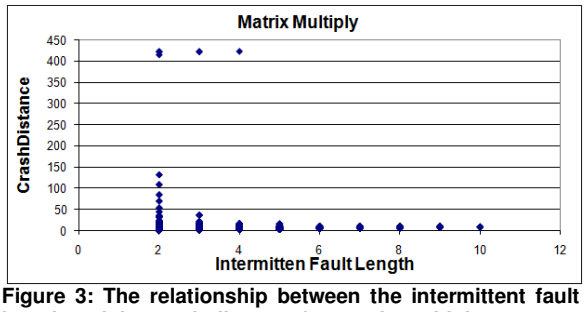

Figure 3: The relationship between the intermittent fault length and the crash distance for matrix multiply.

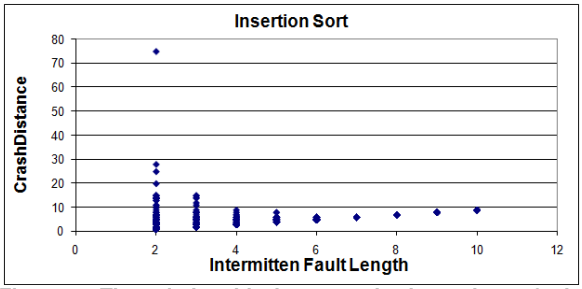

Figure 4: The relationship between the intermittent fault length and the crash distance for Insertion Sort.

Crash distances for intermittent faults of lengths less than 5 nodes are usually larger than those caused by intermittent faults of greater lengths. However, in the next question, we will show that even for faults with large crash distances, only a few dynamic instructions process corrupted data before the crash.

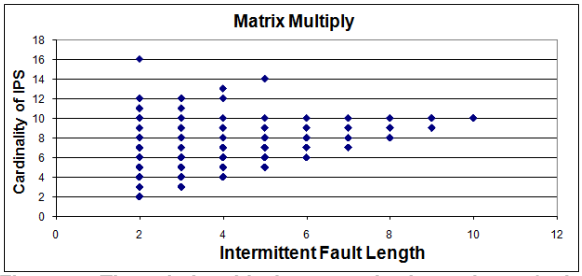

Figure 5: The relationship between the intermittent fault length and the cardinality of IPS for matrix multiply.

Question 3. How many erroneous nodes are generated after the start of the intermittent fault and before the program crashes?

In this question, we study the relationship between the number of erroneous nodes affected by the intermittent faults (i.e. cardinality of IPS, Section 2.3) and the intermittent fault length for both programs (Figure 5 and Figure 6).

We find that the IPS ranges between 2-16 nodes for matrix multiply (Figure 5) and 2-10 nodes for Insertion Sort (Figure 6). *Therefore, the intermittent propagation set does not exceed 16 nodes for intermittent faults of any length.* To understand this phenomenon we consider two extreme cases of intermittent fault lengths:

First, while short intermittent faults (≤ 5 nodes) are likely to propagate beyond the end of the fault, such faults are likely to encounter a crash point in the DDG within at most 16 nodes from the start of the fault. In other words, they have limited propagation before the program crashes.

Second, longer intermittent faults often cause the program to crash even before the end of their occurrence period. We find that the maximum length that an intermittent fault can have before causing the program to crash is 10 instructions. Note that the IPS contains at least the nodes that are directly affected by the intermittent fault, and hence longer faults have a higher IPS cardinality (but still less than 16).

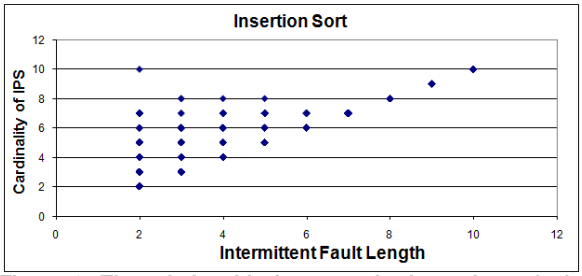

Figure 6: The relationship between the intermittent fault length and the cardinality of IPS for Insertion Sort.

5. Related Work

Fault avoidance and tolerance techniques: Fault-avoidance techniques attempt to mitigate intermittent faults by minimizing process variations [3], voltage regulation [14] or through thermal management [15]. Although such techniques reduce the base rate of intermittent faults, many faults still occur and escape to the software [16]. Fault tolerance techniques [5, 17-18] for intermittent faults require circuit-level changes [17], or often incur very high overheads [5] even for fault-free devices.

Wells et al. [16] propose to recover from intermittent faults by suspending the faulty core and using a virtualization layer to manage over-committed systems. However, they assume that dedicated circuits are used to detect intermittent faults. To the best of our knowledge, there is no technique to mitigate intermittent faults using software alone. In this paper, we evaluate the propagation of intermittent faults in programs as a first step in building software-only mechanisms to tolerate such faults.

Fault propagation studies: Gracia et al. [6] study the behavior of intermittent faults in the VHDL model of a commercial microcontroller. They find that

intermittent fault length is the most influential variable in error propagation. However, they do not consider the impact of intermittent faults on the program executing on the processor, which is important for developing software fault-tolerance mechanisms. While similar studies have been performed for permanent faults [8] and transient errors [7, 9], no such study has been performed for intermittent faults (to the best of our knowledge).

6. Conclusions

In this work, we study the propagation of intermittent hardware faults at the program-level by gathering a dynamic instruction trace of fault-free program execution and analyzing this information using the Dynamic Dependency Graph (DDG). We find that the majority of intermittent faults (~95% for the programs studied) cause programs to crash within few hundreds of dynamic instructions of the fault's start (hence large crash distances are infrequent). Moreover, the number of the dynamic data values corrupted by intermittent faults is at most 16 before the program crashes (if it does) due to the fault.

Our preliminary findings suggest that (1) software-based detection of intermittent faults can be efficient because very few intermittent faults propagate extensively and hence require specialized detection techniques, (2) diagnosis of intermittent faults is feasible using software-based techniques [19] because the propagation sets for intermittent faults is limited to a few dozens of dynamic instructions, and (3) fine-grained check-pointing techniques on the order of thousands of instructions will be effective in recovering from intermittent faults because the crash distance of such faults is less than a few hundred instructions (and hence the fault is unlikely to corrupt a checkpoint).

References

1. Constantinescu, C., *Impact of Intermittent Faults on Nanocomputing Devices*, in *Workshop on Dependable and Secure Nanocomputing*. 2007.
2. McPherson, J.W. *Reliability Challenges for 45nm and Beyond*. in *ACM IEEE Design Automation Conference*. 2006.
3. Borkar, S., et al. *Parameter variations and impact on circuits and microarchitecture*. in *Design Automation Conference*. 2003.
4. Constantinescu, C., *Intermittent faults and effects on reliability of integrated circuits*, in *Reliability and Maintainability Symposium*. 2008. p. 370-374.
5. Smolens, J.C., et al., *Detecting emerging wearout faults*, in *IEEE Workshop on Silicon Errors in Logic - System Effects*. 2007.
6. Gracia, J., et al., *Analysis of the influence of intermittent faults in a microcontroller*, in *IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems*. 2008. p. 1-6.
7. Smolens, J.C., et al. *Fingerprinting: Bounding Soft-Error Detection Latency and Bandwidth* in *Symposium on Architectural Support for Programming Languages and Operating Systems*. 2004.
8. Li, M.-L., et al. *Understanding the Propagation of Hard Errors to Software and Implications for Resilient System Design*. in *International Conference on Architectural Support for Programming Languages and Operating Systems*. 2008.
9. Gu, W., et al. *Characterization of Linux Kernel Behavior under Errors*. in *International Conference on Dependable Systems and Networks*. 2003.
10. Burger, D. and T.M. Austin, *The SimpleScalar tool set, ver-sion 2.0.* Computer Architecture News, 1997. **25**(3).
11. Agrawal, H. and J.R. Horgan, *Dynamic Program Slicing.* ACM SIGPLAN Notices, 1990. **25**(6).
12. Kao, W.-l., R.K. Iyer, and D. Tang, *FINE: A Fault Injection and Monitoring Environment for Tracing the UNIX System Behavior under Faults.* IEEE Transaction on Software Engineering, 1993. **19**(11).
13. Pattabiraman, K., Z. Kalbarczyk, and R.K. Iyer. *Application-Based Metrics for Strategic Placement of Detectors*. in *Pacific Rim International Symposium on Dependable Computing*. 2005.
14. Ernst, D., et al. *Razor: A Low-Power Pipeline Based on Circuit-Level Timing Speculation*. in *International Symposium on Microarchitecture*. 2003.
15. Skadron, K., et al., *Temperature-aware microarchitecture: Modeling and implementation.* ACM Transactions on Architecture and Code Optimization, 2004. **1**(1): p. 94 - 125.
16. Wells, P.M., K. Chakraborty, and G.S. Sohi. *Adapting to intermittent faults in multicore systems*. in *International Conference on Architectural Support for Programming Languages and Operating Systems*. 2008.
17. Ismaeel, A.A. and R. Bhatnagar, *Test for detection & location of intermittent faults in combinational circuits.* IEEE transactions on reliability, 1997. **46**(2): p. 269-274.
18. Blough, D.M., G.F. Sullivan, and G.M. Masson, *Intermittent Fault Diagnosis in Multiprocessor Systems.* IEEE Transactions on Computers, 1992. **41**(11): p. 1430 - 1441.
19. Rashid, L., K. Pattabiraman, and S. Gopalakrishnan, *Formal Diagnosis of Hardware Transient Errors in Programs*, in *IEEE Workshop on Silicon Error in Logic-System Effects*. 2010.