Characterizing the Impact of Intermittent Hardware Faults on Programs

Layali Rashid, *Member, IEEE,* Karthik Pattabiraman, *Member, IEEE,*
and Sathish Gopalakrishnan, *Member, IEEE*

Abstract—Extreme CMOS technology scaling is causing significant concerns in the reliability of computer systems. Intermittent hardware errors are non-deterministic bursts of errors that occur in the same physical location. Recent studies have found that 40% of the processor failures in real-world machines are due to intermittent hardware errors. A study of the effects of intermittent faults on programs is a critical step in building fault-tolerance techniques of reasonable accuracy and cost. In this work, we characterize the impact of intermittent hardware faults in programs using fault-injection campaigns in a microarchitectural processor simulator. We find that 80% of the non-benign intermittent hardware errors activate a hardware trap in the processor, and the remaining 20% cause silent data corruptions. We have also investigated the possibility of using the program state at failure time in software-based diagnosis techniques, and found that much of the erroneous data is intact and can be used to identify the source of the error.

Index Terms—*Intermittent hardware faults, fault propagation, fault diagnosis, fault injection, fault model.*

NOMENCLATURE

t_L	Intermittent fault length
t_A	Intermittent fault active duration
t_I	Intermittent fault inactive duraction

ABBREVIATIONS

CD	Crash Distance
IPS	Intermittent Propagation Set
MNS	Masked Nodes Set
RR	Re-used Registers

I Introduction

Over three decades of continuous shrinking of transistor sizes has led to tremendous improvements in processor performance and at the same time to large challenges in maintaining its reliability. Studies have shown that future processors will be more susceptible to intermittent faults and that the rates of these faults will increase due to technology scaling [1], [2], [3]. A recent study has found that about 40% of real-world failures in a processor are caused by intermittent faults [4]. Intermittent hardware errors are bursts of non-deterministic errors that occur at the same location (i.e., microarchitectural component) [2]. They can occur due to variations in the manufacturing process [5], in-progress wearout and manufacturing residues [1], [2], [6], [7].

Fault avoidance techniques mitigate intermittent faults by minimizing process variations [1], regulating voltage [8] or managing temperature [9]. Although such techniques reduce the base rate of intermittent faults, many such faults still occur and escape to the software [3]. Therefore, we need mechanisms at the software level to mitigate the impact of intermittent faults through detection, diagnosis and recovery.

Intermittent faults lie between the extremes of transient and permanent faults. Transient faults are one-time events that usually last for one cycle and are unlikely to recur in the same location. Transient faults are mainly caused by alpha particles from packaging material, cosmic rays or thermal neutrons. Permanent faults are caused by reproducible and irreversible hardware defects that occur indefinitely in the same microarchitectural location. Permanent faults are mainly caused by transistor wearout due to persistent stress conditions such as extreme temperature. Prior work has analyzed the effects of transient faults [10], [11], [12] and permanent faults [13], [14] on programs. These analyses played a vital role in designing fault-tolerance techniques that mitigate the effects of errors at low performance and area overheads.

However, intermittent faults are unlike transient faults in that they recur at the same location and are unlike permanent faults in that they do not persist indefinitely, but rather occur non-deterministically. Therefore, it is not straightforward to apply studies and techniques that have been designed for permanent and transient faults to intermittent ones.

In this work, we investigate the main characteristics of intermittent faults and the implications of such characteristics on error detection, diagnosis and recovery. Further, we study the impact of intermittent faults on programs as a first step towards building software-level mechanisms for isolating the source of intermittent faults (see Section VII for more details).*To the best of our knowledge, ours is the first comprehensive study of intermittent*

faults using fault injections and a realistic set of benchmarks.

A hardware fault can affect programs in different ways. The fault may be benign (changes to program state are benign), cause silent-data corruption (changes to program state are erroneous) or lead to a crash (i.e., hardware trap). Even if an error does lead to a program crash, the crash may not occur immediately after the onset of the error, but only after some amount of time later. In the meantime, the error may propagate and corrupt the program state even more. It is important, in this context, to ask: (1) What is the fraction of intermittent faults that lead to program crashes? (2) For the faults that do lead to crashes, how much do they propagate within their programs before the crashes? (3) How useful is the program state at failure time for the software-based fault-diagnosis techniques?

In this study, we focus on faults that change program state erroneously and hence ignore inactivated faults and faults that are activated but are benign. Inactivated and benign faults may occur in infrequently used locations or locations of little importance to the program. Therefore, these faults are likely to have little impact on the processor's reliability. Further, we focus on errors that lead to program crashes in this study because our experiments indicate that more than 80% of non-benign intermittent errors result in program crashes. Finally, we focus on intermittent error diagnosis rather than detection and recovery because most diagnosis techniques for hardware errors assume that the error is reproducible, which does not apply to intermittent errors.

We answer the questions asked above by subjecting a few SPEC2006 benchmark programs to a fault-injection campaign at the microarchitecture level. Ideally, faults are injected at the logic-level simulations or are triggered by stressing a physical processor. However, the goal of our study is to characterize the effects of intermittent faults on real benchmarks, and the use of such environments is prohibitively expensive and provides limited observability in our experiments.

We build our fault-injection tool on the top of a microarchitectural simulator. Our tool can inject intermittent faults into a variety of microarchitectural locations, with different lengths, starting times and models. Moreover, our tool facilitates a set of analysis on programs by supporting detailed program tracing. The contributions of our work are as follows:

1) Building a fault-injection tool at the microarchitectural level to inject intermittent faults into programs.
2) Characterizing the impact of intermittent faults on programs through fault-injection experiments using the SPEC2006 benchmarks.
3) Studying the relationships between intermittent-faults parameters such as location and model on the severity of the fault consequences.
4) Projecting the results of our study on intermittent-errors detection, diagnosis and recovery (Section VII).

The main results of the study are as follows:

- Between 41% and 63% (varied by benchmark) of the faults we injected led to program crashes. Of the remaining fault injections, between 22% and 43% were benign, and only about 14% of injected faults resulted in silent-data corruption (SDCs).
- 96% of the crash-causing errors lead to program crash within one hundred thousand dynamic instructions.
- 87% of the crash-causing errors corrupt less than 500 data values before program crash.
- 42% of the corrupted data are not masked by other correct data at program crash.

II Related Work

A survey of previous work on hardware faults characterization and tolerance is provided in this section. We find that research in intermittent errors characterization and tolerance techniques is still in its infancy. We also find that software-based techniques are promising, hence this work is directed toward showing their potential.

II.1 Fault-Characterization Studies

1. Intermittent faults: Pan et al. [15] proposed the Intermittent Vulnerability Factor (IVF) to characterize the sensitivities of different microarchitectural units to intermittent faults. IVF is based on the Architectural Vulnerability Factor (AVF) metric, which is an analytical measure of the vulnerability of microarchitectural units to soft errors [16]. However, Wang et al. [17] found that the AVF technique overestimates the vulnerability of units to soft errors by two to three times compared to an equivalent fault-injection experiment. Such overestimates can lead to vastly more conservative designs than necessary, thus incurring unnecessary power and performance overheads. Therefore, we use a fault injection based approach to study the impact of intermittent faults on processors.

Gracia et al. [18] studied the behavior of intermittent faults in a VHDL model of a commercial microcontroller using Bubblesort algorithm. They found that intermittent fault length[1] is the most influential variable in error propagation. However, they did not consider the impact of intermittent faults on programs executing on the processor, which is important for developing fault-tolerance mechanisms.

In our previous work we performed a preliminary study of intermittent faults using functional simulation [19]. We focused on small benchmarks and a very restricted fault model that represent intermittent faults as bit-flips. Moreover, our study in [19] did not explore the impact of fault masking on software-based diagnosis techniques. In a follow-up work, we compared the effect of transient and intermittent faults on programs [20]. As in the prior study, we characterized small benchmarks

1. Fault length is the full duration of the fault or t_L in Figure 1.

on a microarchitectural simulator using a restricted fault model.

2. Permanent faults: Li et al. [13] performed a study similar to ours for permanent faults. They found that 95% of the injected faults can be detected by monitoring for a hardware trap, hang or excess operating system calls. The remaining 5% are benign faults that do not affect the program state. They also found that 86% of the detected faults can be detected within 100K instructions.

Karimi et al. [14] injected transient and permanent faults into the control logic of an RTL simulator augmented with a functional simulator. Although their simulator supports arbitrary fault-starting points and lengths they did not consider intermittent faults.

3. Transient faults: The effect of transient faults on programs is a well-studied topic [10], [11], [12]. For example, Gu et al. [10] conducted massive fault-injection campaigns into Linux-kernel to study the impact of transient faults. They found that programs do not necessarily crash immediately but often continue executing as the error propagates. They also show that most transient errors cause a crash within 10 cycles. Further, other work has evaluated the effects of hardware faults on software [21], [22]. For example, Huang et al. found that the impact of a hardware fault depends on the fault location and time and is software specific.

Summary: Unlike permanent faults, intermittent faults do not persist indefinitely but rather occur non-deterministically. Moreover, unlike transient faults, intermittent faults tend to recur in the same location. Therefore, the results of permanent or transient-faults characterization cannot be generalized to intermittent faults and a new characterization study is required.

II.2 Fault-Tolerance Techniques

Current fault-diagnosis techniques for intermittent faults require circuit-level changes [23], special hardware recorders [24], [25], or periodic testing [7] (such periodic tests cover only a class of intermittent faults that manifest themselves under reduced frequency guardbands). All these techniques impose high area or performance overheads even for fault-free devices. Other mechanisms such as SWAT by Li et al. [26], and the work by Pellegrini and Bertacco [27] are invoked only upon fault detection (hence, do not result in performance overhead for fault-free cores). Li et al. [26] diagnosed hardware faults by monitoring the application for abnormal events and initiating error diagnosis only upon error detection. Pellegrini and Bertacco [27] technique monitored the usage of the processor's functional units and ran tests only for the units used by the application. However, both works focus on permanent hardware faults and assume that the fault occurs during the testing period. This assumption does not apply to intermittent faults.

As for intermittent error recovery, Wells et al. [3] proposed to recover from intermittent errors in software by suspending the faulty core and using a virtualization layer to manage over-committed systems. They assumed that hardware circuits are used to detect intermittent errors. These circuits incur power and area overheads, and are not suitable for commodity systems. In our previous work [28] we evaluated the impact of different intermittent error recovery scenarios on the processor performance. These scenarios describe different possible actions that can be taken when an intermittent error is detected or if the error leads to a crash. The scenarios are: (1) restoring program state to the last checkpoint only, (2) restoring to the last checkpoint in addition to disabling the faulty microarchitectural unit or (3) restoring to the last checkpoint in addition to disabling the faulty core. We found that disabling the faulty microarchitectural unit results in better performance if the intermittent error has high frequency, and if it occurs in non-critical units.

The work in this paper is geared towards understanding the propagation of intermittent faults in software programs, and thus design efficient fault tolerance mechanisms for them. To our knowledge, there has been no prior work on software mechanisms for intermittent fault tolerance.

III Intermittent Faults: Background, Modeling and Fault Injection

In this section, we define intermittent faults, discuss their root causes and rates of occurrence (Section III.I). We then propose intermittent fault models at the microarchitecture level (Section III.II). Finally, we build our fault injection tool based on the microarchitectural fault models (Section III.III).

III.1 Background

Definition We define an intermittent fault[2] as one that appears non-deterministically at the same hardware location, and lasts for one or more (but finite number of) clock cycles. This is consistent with definitions in prior work [2], [3]. The main characteristic of intermittent faults that distinguishes them from transient faults is that they occur repeatedly at the same location, and are caused by an underlying hardware defect rather than a one-time event such as a particle strike. However, intermittent faults appear non-deterministically (unlike permanent faults) and only under extreme operating conditions. Similar to Gracia et al. [18], we characterize an intermittent fault using the following parameters:

- Fault length: the full duration of a fault (t_L).
- Fault active duration: the duration at which the fault manifests itself to the instructions that use the defective hardware part (t_A).

2. We follow the standard fault, error, failure terminology in Avizienis et al. [29]. A fault is the physical defect in the device/circuit, an error is the corruption of program data due to the fault, and a failure is the program crashing, generating erroneous results or hanging as a result of the error.

- Fault inactive duration: the duration at which the fault does not manifest itself to the instructions that use the defective hardware part (t_I).
- Fault location: the physical location of the fault. It consists of a microarchitectural unit and a bit.
- Fault model: a representation of the fault at the microarchitecture level. Different models represent different root causes (Section III.II).

Figure 1 shows an example of an intermittent fault with the above parameters.

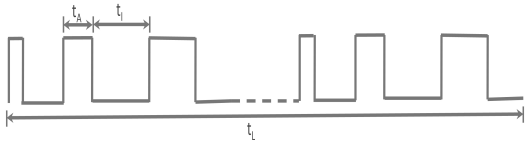

Fig. 1: The fault model used in this work. t_L is the fault length, t_A is the fault active duration and t_I is the fault inactive duration.

Causes The major cause of intermittent faults is device wearout, i.e., the tendency of solid-state devices to degrade with time and stress. Wearout can be accelerated by aggressive transistor scaling which makes processors more susceptible to extreme operating condition such as voltage and temperature fluctuations [1], [5]. Many well studied transistor failure mechanisms such as dielectric breakdown, negative bias temperature instability (NBTI), hot-carrier injection (HCI) and electromigration [5] occur due to wearout. In-progress wearout faults are typically intermittent as they depend on the operating conditions and the circuit inputs. In the long term, these faults may eventually lead to permanent defects.

Another cause of intermittent faults is manufacturing defects that escape VLSI testing [2]. Often, deterministic defects are flushed out during this testing and the ones that escape are non-deterministic defects, which emerge as intermittent faults.

Finally, design defects can also lead to intermittent faults, especially if the defect is triggered under rare scenarios or conditions [30]. We do not model design defects because they do not usually follow specific patterns, i.e., each defect affects the processor in a different way.

Table 1 summarizes the major causes of intermittent faults that we consider in this study.

Occurrence Rates Few field studies have been conducted to monitor intermittent-faults rates. Constantinescu [2] found that 6.2% of the hardware errors in memory subsystems are intermittent. However, he did not present any data for processor faults. A recent study by Nightingale et al. [4] analyzed error logs sent by Microsoft Windows Error Reporting program from 950,000 personal computers. They found that, of the hardware errors reported about microprocessors, approximately 39% are intermittent. However, they only consider a small class of processor errors, and only those that cause operating-system crashes. As a result, their study is limited to certain types of intermittent errors that occur in the field.

III.2 Microarchitectural Fault Models

We rely on prior work to build approximate intermittent fault models at the microarchitecture level. The intermittent faults considered in this work are caused by temperature/voltage fluctuations, wearout and manufacturing defects. Moreover, the focus of this work is on faults that occur in the processor. We do not consider errors that occur in the memory and input/output hierarchy. Memory and input/output hierarchy errors are usually tolerated by Error Correcting Codes (ECC) and Cyclic Redundancy Checks (CRCs).

Modelling intermittent faults at the microarchitectural level is challenging because we need to abstract the effects of the faults to the microarchitecture. Unfortunately, to the best of our knowledge, there has not been prior work in this area. Most prior work that model in-progress wearout faults, do so at the RTL-level or gate-levels, where faults manifest themselves as delays in transistor-switching times [35]. However, the need to run real workloads and monitor the execution for large number of cycles makes it impractical for us to use these low-level simulators. Therefore, we came up with a microarchitectural fault model in Table 2. We make the following assumptions in the modeling:

- At any time, a microarchitectural unit may be affected by at most a single intermittent fault. Also at any time, at most a single microarchitectural unit may be affected by an intermittent fault. These two assumptions are justified because intermittent faults are relatively infrequent compared to the typical execution time of many applications.
- In-progress wearout effects (Dielectric breakdown, NBTI, HCI and electromigration) manifest themselves as stuck-at last value (Smolens et al. [7]). We assume that the delay caused by the fault is longer than a clock period; otherwise it will not affect the output of a transistor.
- Electromigration faults also manifest themselves as intermittent stuck-at zero/one and dominant 0/1. Although these faults occur in interconnects, we assume that they will ultimately lead to similar faults in the register destinations of the interconnects.
- Manufacturing defects manifest as intermittent stuck-at zero/one and dominant 0/1 (Gracia et al. [18]).

III.3 Fault Injection Tool

We couple our fault-injection with a simulation environment. A key challenge in our study is to find a simulation environment that is (1) feasible (simulation finishes in reasonable time), (2) accurate (good representation of the actual faults), (3) observable (to monitor the internal program and processor states) and (4)

TABLE 1: Description of different intermittent failure mechanisms.

Failure Cause	**Description**
Dielectric breakdown	Thin gate oxide results in the gate becoming more vulnerable to high voltages. Manufacturing traps in these gates get invariably charged as current flows in the oxide. These charged traps start conducting large currents with time. This current leads to thermal damage to the transistor and creates more traps. Over time, the traps accumulate and create a conductive path between the transistor metal and substrate. This path increases leakage current in the transistor and leads to soft gate-oxide breakdown. With persistent stress conditions more traps formulate, and over time this conductive path becomes a cross section throughout the gate and connects current from the substrate to the metal. This is called hard gate-oxide breakdown [31].
Negative bias temperature instability	Thin gate oxide and high temperatures may result in the silicon-hydrogen bonds in the interface between the gate oxide and the substrate to break. This releases hydrogen ions and the vacant positions in the gate can formulate holes. These holes change the transistor characteristics by increasing the threshold voltage, therefore transistor will get slower over time [32].
Hot carrier injection	Short transistor channel lengths increase the transistor's speed but at the same time increase the electrical field in the channel. With high voltages, this may lead to some electrons or holes gaining enough kinetic energy to get injected from the substrate into the gate oxide. This leads to a degradation in the transistor's threshold voltage [33].
Electromigration	Small wire geometry and high electrical fields may lead the metal atoms to migrate from one place (which may lead to open circuit) and pile up in other places (which may lead to short circuit with other wires) [34].
Manufacturing defects	This includes process, voltage and temperature variations (PVT) [1] and residues in cells [2].

TABLE 2: Intermittent faults' causes and models.

Fault mechanism	**Causes**	**Gate-level models**	**Microarchitectural models**
Dielectric breakdown	Infant-mortality Thin gate oxide High voltage	Intermittent delay	Intermittent stuck-at-last-value
Negative bias temperature instability	Thin gate oxide High temperature	Intermittent delay	Intermittent stuck-at-last-value
Hot carrier injection	Short channel length High Voltage	Intermittent delay	Intermittent stuck-at-last-value
Electromigration	Small wires geometry High temperature High current density	Intermittent delay Intermittent open Intermittent short	Intermittent stuck-at-last-value Intermittent stuck-at-zero/one Dominant-0/1 bridging
Manufacturing defects	—	Intermittent open Intermittent short	Intermittent stuck-at-zero/one Dominant-0/1 bridging

repeatable (to perform replays). In prior work, logic-level or RTL-level simulations have been used to study intermittent faults [7], [18]. However, due to the very detailed simulations, only very small programs can be used in these simulations. Since the goal of our study is to characterize real benchmarks (SPEC2006), we use a relatively fast microarchitectural simulation. At the same time, we implement approximate fault models to represent intermittent faults at the microarchitecture level (see Table 2 for more details about our fault model).

We build our fault-injection tool based on the SimpleScalar simulator (Alpha sim-outorder) [36], [37], which is a cycle-accurate simulator. This is a detailed microarchitectural simulator with branch predictor, caches and external memory. It models an Alpha processor, a RISC ISA. We choose this simulator because it models the majority of the microarchitectural components of interest to our study and is widely used in the computer architecture community. In addition to the features available in SimpleScalar, we have added the following functionalities:

Inject intermittent hardware faults at the microarchitecture level. The user can specify the following parameters for every fault:

1) Fault start and end cycle.
2) Fault location, which consists of a combination of a microarchitectural component and a bit position within the component's output. The available components are (a) the fetch unit, both the fetched instruction and the PC, (b) the destination register of the integer ALU, multiplier and divider, the destination register of the FP adder, multiplier, divider, comparator, square root, and FP-to-integer converter, (c) the load-store unit read address, write address and data, and (d) the load-store queue data and address.
3) Fault activity and inactivity durations.
4) Fault models which include Stuck-at-one/zero/last-value and Dominant-0/1.

Emulate hardware traps. The implemented traps are divide by zero, data overflow, invalid instruction, invalid PC, invalid memory address and invalid memory alignment. This is required for accurate accounting of number of program crashes. Basic SimpleScalar does not have this feature.

Record crash dump file in the event of a crash. This file includes the contents of the register file, memory footprint and PC at crash time, hardware-trap type and crash distance[3]. Our simulator also records a detailed trace of the running program. The trace consists of a list of executed instructions, with the following information recorded for each instruction: instruction type (unary, binary, jump, branch, load or store), PC, input registers values and output registers values.

Replay the execution and analyze the effects of an intermittent error that caused a crash by using a faulty run and the corresponding replay. The output of this analysis includes: (1) propagation set, (2) masked nodes set (MNS) and (3) reused registers (RR)[4]. Moreover, this analysis can distinguish between a crash (activation of a hardware trap), silent data corruptions (erroneous data in registers and memory when program finishes), and benign runs (injected faults with no erroneous data).

3. Crash distance is the number of dynamic instructions that execute from the start of the error until program crashes.

IV Methodology

In this section, we first define metrics to quantify the impact of intermittent faults in programs (Section IV.I). We then demonstrate the metrics through an example (Section IV.II).

IV.1 Terms and Definitions

A node is a value produced by a dynamic instruction during program execution. A node can be a data value or a memory address. Further, a node can be read multiple times but is written only once during execution.
A faulty run is a program execution during which an intermittent fault is injected, and activated (i.e., read by the program).
A replay is a re-run of the instructions that executed during the faulty run. No faults are injected during the replay.
A crash node is the node at which a program crashes due to an intermittent error.
The Crash Distance or CD is the number of nodes that are generated by a program from the start of an intermittent fault until a crash node is reached.
The Intermittent Propagation Set or IPS is the set of nodes to which an intermittent error propagates until a crash node is reached.
The Masked Nodes Set or MNS is the set of nodes to which an intermittent error propagates until a crash node is reached. However, these nodes contain correct values because of error masking. Error masking can happen due to (1) logical instructions, such as AND/OR instructions, (2) test and set instructions, such as branch if less than zero instruction (BLTZ) or (3) the fault model does not change the bit value, e.g. a stuck-at-zero in a bit whose value is zero.
MNS can be found by comparing the node values in the IPS in the faulty run with the corresponding values in the replay. If an erroneous node has the same value in faulty run as that in the replay, then the erroneous value has been masked.
The Faulty Registers is the set of registers that have erroneous values when the program crashes.
The Re-used Registers or RR is the number of registers that had erroneous values but were overwritten by correct values before a crash occurs. RR can be found by $RR = |Registers(IPS) - Registers(MNS) - Registers(FaultyRegisters)|$, where $Registers(s)$ means

4. More details are available in Section IV.I.

the set of destination registers used to store s set of nodes.

IV.2 Example

We demonstrate how to compute the CD, IPS, MNS and RR of an intermittent error using an example code fragment shown in Table 3.

The second column of the table shows the node values that result from executing the corresponding instruction in a faulty run (the run affected by the error). The third column shows the node values that result from executing the corresponding instruction during the replay run (error-free run). The last column shows the mapping from instructions to nodes. 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.

In the example code fragment, elements at indices 15 and 45 of an array (starting at $Array\_Addr$) are loaded into registers R4 and R5, respectively (node 1, 2, 4, 5). Then, these two registers are added and the result is stored in R6 (node 6). An immediate value of 77 is stored in R3 (node 3). The result in R6 is then logically ANDed with R3 (node 7). Next, an immediate value of 23 is stored in R1 (node 8). Finally, R6, which is an index in the same array, is used to load the corresponding array-item to R2.

Assume that an intermittent stuck-at-zero error affects the third bit in the destination register of the first three instructions in Table 3. This error results in a crash at node 9 due to invalid memory address. We want to compute CD, IPS, MNS and RR of this error.

Recall that IPS is the set of nodes to which an intermittent error propagates. Next, we will follow the error propagation for each of the first three erroneous instructions and add the erroneous nodes that result from the error propagation to a set that will form IPS. As for the error that affects node 1, it includes at least the node at which the error occurs, namely node 1. Node 1 has a successor node 4, node 4 is used to compute node 6. Node 6 is, in turn, used to compute nodes 7 and 9. Hence, IPS so far is $\{1, 4, 6, 7, 9\}$. Similarly, we can follow the propagation of node 2 and find that it manifests the error to nodes $\{2, 5, 6, 7, 9\}$. While node 3 propagates to nodes $\{3, 7\}$. Therefore, $IPS = \{1, 2, 3, 4, 5, 6, 7, 9\}$.

As for CD, it is the number of nodes generated from the start of the intermittent fault at node 1 until the crash node 9, i.e., $CD = 8$.

To find the MNS, we compare the values of all nodes that appear in IPS with the corresponding nodes in the replay. We find that all nodes have different values except for node 7, which has the same value in the faulty run and in the replay run. Hence, the error has been masked, by logical AND instruction, and $MNS = \{7\}$.

To find the RR, we check the destination registers for nodes that appear in IPS. We find that according to IPS, R1-7 should include erroneous data. Next, we check MNS and notice that R7 (node 7) has a masked error, therefore, R1-6 should include erroneous data. However, R1 is not in the faulty registers (registers whose values are erroneous at crash time). Formally, $RR = |\{1, 2, 3, 4, 5, 6, 7\} - \{7\} - \{2, 3, 4, 5, 6\}| = 1$. Therefore, $RR = 1$.

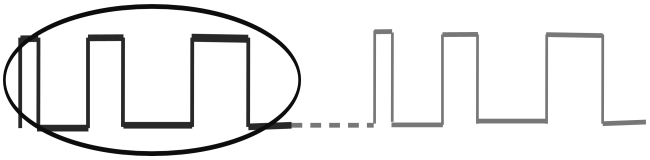

Fig. 2: The fault model used in characterizing faults. We focus on one set of active and inactive durations that occur within seconds.

TABLE 3: Code fragment to illustrate IPS, CD, MNS and RR computation.

Code fragment	Node value at faulty run	Node value at replay	Node No.
mov R1, #15	11	15	1
mov R2, #45	41	45	2
mov R3, #77	73	77	3
ld R4, R1, Array_Addr	681	10	4
ld R5, R2, Array_Addr	504	9	5
add R6, R5, R4	1185	19	6
and R7, R6, R3	1	1	7
mov R1, #23	23	23	8
ld R2, R6, Array_Addr	CRASH	950	9

V Experimental Setup

We used fault injections at the microarchitecture level to characterize intermittent faults. In this section, we describe our experimental setup.

Fault Parameters We follow the fault model described in the previous section (Section III). However, at the microarchitecture level, we cannot model the recurrence of intermittent faults in hours or days because it is prohibitively expensive to do so. Rather we focus on one set of active and inactive durations that occur within seconds (circled and in bold font in Figure 2). As a result, we choose the parameters in our experiments to capture a single burst.

Table 4 shows the parameter values we use in our experiments. Each fault-injection experiment involves choosing a fault parameter in the specified range based on uniform distribution. Note that due to the limited information known about intermittent errors characteristics, we were not able to find the exact active and inactive duration. Therefore, we experiment with many

active/inactive durations that range from 5 cycles to 20,000 cycles. We chose these numbers because voltage fluctuations last from 5 to 30 cycles [38], while temperature fluctuations may last hundreds of thousands of cycles [9] (voltage and temperature fluctuations are leading causes of intermittent faults).

Benchmarks: We used 7 integer and 4 FP benchmarks from the SPEC CPU 2006 suite for our evaluation. We could not compile the rest of SPEC2006 benchmarks to run in our Alpha simulator, and hence did not use them. We did not cherry-pick these benchmarks based on our results.

Experiment: We used our fault injection tool described in Section III.III to conduct our experiments. Our procedure to perform fault characterization has two steps:

(1) Running benchmarks: For each experiment we ran a benchmark twice. In the first run we injected an intermittent fault (faulty run), while in the second run we did not inject faults, but rather replayed the first run with the same instruction stream (replay run). For both runs, the tool collected program traces and the most recent contents of the register file and memory footprint.

For each benchmark program we injected 3000 faults. We computed confidence bounds on the results with 95% confidence. Only one fault is injected in each execution to ensure controllability. We used the standard SimpleScalar simulator's parameters for the processor and memory configuration (Table 5).

Each benchmark was forwarded for 2 billion instructions to remove initialization effects. Then, an intermittent fault is injected. After the injection, the tool ran the benchmark for 1 million instructions[5]. We did not run benchmarks to completion (as this is a very time-consuming process in our tool). We rather relied on analyzing the trace files of the faulty run and the corresponding replay to find if a fault results in SDC.

(2) Analyzing runs: Our tool compares and analyzes trace files, register files and memory footprints collected in the first step. The purpose of this analysis is to find how the run terminates (SDC, benign or crash). In case of a crash the tool finds CD, IPS, MNS and RR values for the run.

TABLE 4: Fault-injection parameters.

Fault Parameter	Value/Range
Location-bit	A bit position chosen randomly from 0 to 63 in a microarchitetural unit.
Location-unit	Fetch Unit (instruction, PC), integer ALU, multiplier, divider, LSU (data, read address, write address), FPU and LSQ (data, address).
Start cycle	A cycle chosen randomly from 1 to 1,000,000 cycle.
Length (t_L)	5, 50, 1000, 50,000, 100,000, 500,000 or 1,000,000 cycle.
Activity duration (t_A)	5, 50, 100, 500, 10,000 or 20,000 cycle.
Inactivity duration (t_I)	5, 50, 100, 500, 10,000 or 20,000 cycle.
Microarchitectural Model	Stuck-at-one/zero/last-value and Dominant-0/1.

TABLE 5: Simulator configuration parameters.

Configuration Parameter	Value
ALUs/Multipliers/Dividers/FPUs	1 from each type
Fetch/decode/execute/commit rate	1 per cycle
Branch prediction type	perfect
Register update unit size	16
Load-store queue size	8
Register file	32 integer regs, 32 FP regs
Instruction/Data L1	16KB each
L1 hit latency	1 clock cycle
L2 (Unified)	256KB
L2 hit/miss latency	6/18 clock cycles

VI Results

In this section, we present the results of characterizing intermittent faults on programs by fault injection. We first study how a program is affected by an intermittent fault upon termination, by classifying the terminations into crashes, SDCs and benign terminations. Since program crashes are the most common effect of an error, we dig more into this category by finding the CD and IPS for the crash-causing errors (Section VI.I).

We then attempt to answer the question of whether the erroneous data in a program are partially erased by correct data at the time of the crash (note that if erroneous data is completely erased then there will be no crash). This information is critical for any software-based technique that diagnoses an error by analyzing the program crash dump file. This factor is characterized by the MNS and RR (Section VI.II).

Finally, we study the effect of the various intermittent fault parameters on the severity of the error consequences (Section VI.III). These parameters are fault length, fault model and fault location (Section III.I).

VI.1 Impact of Intermittent Faults on How Programs Terminate, CDs and IPSs

In this experiment we injected 3000 faults in each benchmark, one at a time, using the parameters in

5. This number is reasonable because we find that only 2% of the crash-causing faults have a crash distance of more than one million dynamic instructions (Section VI).

Table 4.

Program terminations: We find that 73% of the injected faults are activated (not shown in figures). An activated fault means that the faulty location was used by the program. Inactivated faults happen due to faults injected in bits that are not used or faults that do not change the bit's value. We do not consider inactivated faults in computing the percentages below - this is standard practice in fault injection studies. Note that other studies of transient faults [10] and permanent faults [13] have also found that many faults are not activated.

Out of the activated faults (Figure 3), an average of 53% of the errors lead to program crash, while 14% lead to SDC and 33% of the faults were benign (i.e., had no effect on the program). In other words, of the intermittent faults that are non-benign (i.e., 67% of total faults), 79% result in a program crash. *Thus, a program crash is the dominant effect of activated intermittent faults*. This motivates us to focus on crash-causing faults for the rest of this section.

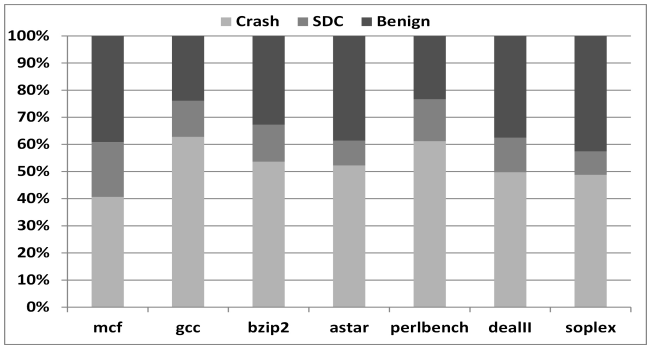

Fig. 3: Intermittent fault impact on how programs terminate. The results are obtained with confidence interval of 95% and error margin of ±4% for crashes, ±2% for SDCs and ±4%for benign runs.

Crash distances: Our results show (Figure 4) that an average of 50% of the injected faults crash within 100 dynamic instructions from the start of the error, 22% crash between 100 and 1,000 dynamic instructions, 24% between 1,000 and 100,000 dynamic instructions, 2% crash between 100,000 and 1,000,000 dynamic instructions and the remaining 2% crash after 1,000,000 dynamic instructions. *These results show that the dominant effect of an intermittent error is to cause a program crash soon after the error occurrence.*

Cardinality of the Intermittent Propagation Set (IPS): The IPS cardinality corresponds to the number of potentially corrupted nodes in the program, which in turn points to the severity of the failure. We plot the cardinalities of the IPS in Figure 5. We find that an average of 68% of the crashes have IPS cardinalities of less than 100 nodes, 19% have IPS cardinalities between 100-500 nodes, 5% have IPS cardinalities between 500-1,000 nodes and the remaining 8% crash after corrupting more than 1,000 nodes in the program. Therefore, the majority of the crash-causing intermittent errors cause limited change to the internal state of program. This is because of their relatively short crash distances as can be seen above.

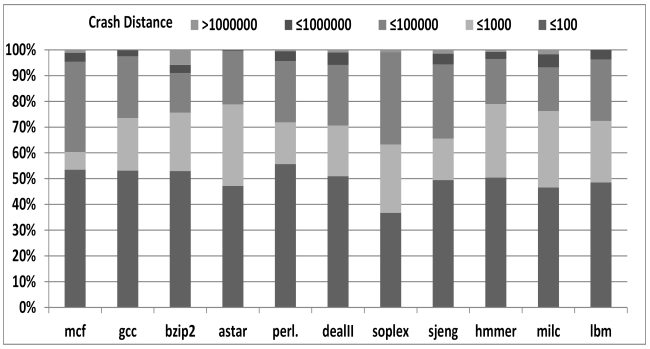

Fig. 4: Crash-distance ranges for crash-causing-intermittent faults.

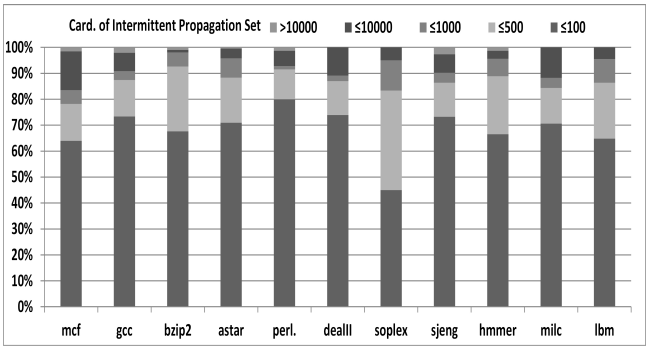

Fig. 5: IPS-cardinality ranges for crash-causing-intermittent faults.

Summary: We find that the majority of intermittent errors that are activated, cause the program to crash. However, most program crashes take place soon after the error occurrence and there is limited error propagation before the crash. The impact of this behavior on fault tolerance techniques in discussed in Section VII.

VI.2 Impact of Error Masking

As we show in Section VII, software-based fault diagnosis techniques rely (completely or partially) on the failure dump or the state of the program at the time of crash/error detection to learn about what caused the error and to find the best way to recover from that error[6]. We refer to the traces left by the error on the program state as "clues". In this subsection, we quantify how many clues are erased prior to the program crash due to masking (MNS) and re-using registers (RR). Note that RR and MNS do not include the faulty node itself - refer to Section IV.I for more details.

6. Software-based diagnosis techniques can also collect data about programs during program execution, we do not cover dynamically collected data as it is technique dependent.

Re-used registers: We study the number of re-used registers at the time the program crashes, regardless of whether the error propagated other registers before the register was overwritten. For example, if a faulty register #3 is used to update register #12 before it is overwritten with correct data, then we count it as re-used register. The results are shown in Figure 6.

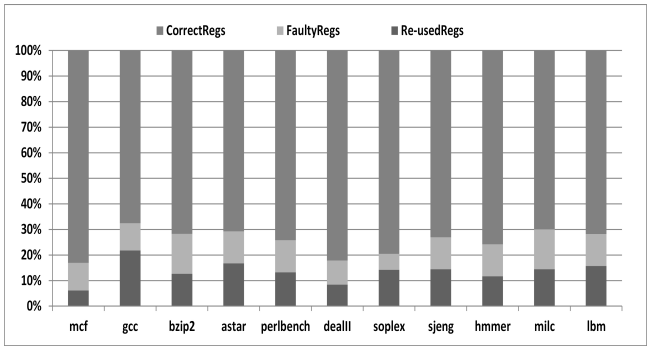

Fig. 6: Distribution of number of correct, faulty and re-used registers at crash time. The results are obtained with confidence interval of 95% and error margin of ±3% for correct registers, ±2% for re-used registers and ±2% for faulty registers.

We find that, on average, 14% of registers are re-used, 12% of the registers are faulty because of error propagation, and the remaining 74% of the registers are not modified by the error. Therefore, out of the registers that store erroneous data at some point during program execution, 54% are overwritten with correct data, and the remaining 46% have erroneous data at crash time. These results can be explained by the short crash distances observed in Figure 4, which implies that programs affected by intermittent errors do not execute long enough after the error occurrence to overwrite a large number of registers.

Masked nodes set: On average, 58% of erroneous nodes are masked and the remaining 42% of nodes will contain erroneous data at crash time (Figure 7). As we mentioned in Section IV.I, nodes masking occurs due to (1) logical instructions, such as AND/OR instructions, (2) test and set instructions, such as branch if less than zero instruction (BLTZ) or (3) the fault model does not change the bit value, e.g. a stuck-at-zero in a bit whose value is zero. Although the majority of the erroneous nodes are masked by the time the program crashes, the absolute number of nodes that are not masked in any run is 60, on average. Therefore, with careful design, software-based diagnosis technique may still be able to use erroneous data to tolerate faults with reasonable accuracy.

Summary: We find that 46% of the erroneous registers are not overwritten with correct data, and 42% of the erroneous nodes are not masked. Therefore, much of the data corrupted in a crashed program is intact and software-based diagnosis approaches are feasible. Nevertheless, these approaches should be designed carefully to accommodate for the lost "error clues" on the program final state before crash.

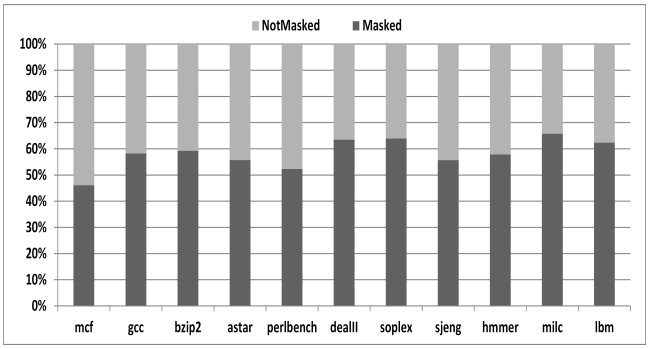

Fig. 7: Distribution of number of masked and non-masked nodes at crash time. The results are obtained with confidence interval of 95% and error margin of ±3%.

VI.3 Impact of Intermittent Fault Properties on the Severity of the Faults

In this subsection, we study the relationships between the intermittent fault length, model and location on the way a program terminates. Since the observations are similar across all benchmarks, in this section we focus on one SPEC2006 benchmark, namely *astar*, to illustrate these differences.

Fault length: As for the fault length impact (Figure 8), we plot the same data that we collected in the previous subsection but we classify it according to the fault length. We note that short faults of 50 cycles lead to small percentage of crashes (34%). The percentage of crashes increases with longer errors until it reaches a threshold of 60% at error length of 50,000 and does not increase afterwards. This implies that the longer the error, the more nodes that are affected by it and the sooner the program crashes until the error length reaches a threshold. After the threshold an intermittent fault behaves more like a permanent fault irrespective of how much longer the fault lasts. We find that the increase in crashes saturated beyond a certain error length due to (1) errors injected into less critical locations, such as infrequently used entries in the load-store queue and (2) error models that are less likely to change the injected bit (e.g., stuck-at-zero).

Fault model: In Figure 9, we plot the same data we collected in the previous subsection, but this time we classify it according to the fault model used. Our data show how the percentages of crashes, SDCs and benign runs vary with the fault model. We do not see a significant difference in number of crashes across the different models, except in two cases. The stuck-at-one and last-value models have relatively higher percentage of crashes (67%). This is because these two models are

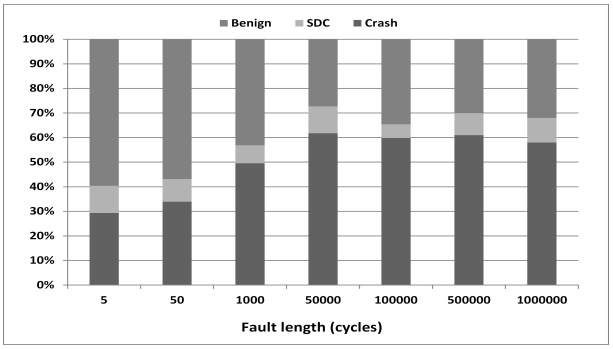

Fig. 8: Effect of intermittent fault length on how programs terminate.

more likely to change the injected bit. Note that unused bits have zero by default. Therefore, the stuck-at-one fault model will flip the unused bit, while stuck-at-last value will prevent updates to the bit if it is either zero or one.

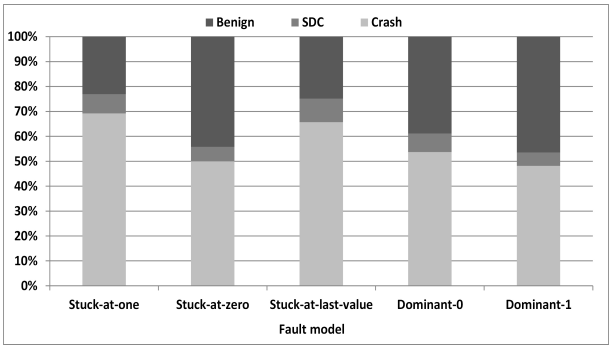

Fig. 9: Effect of intermittent fault model on how programs terminate.

Fault location: We classify the fault injection data we collected in the previous sub-section according to the fault location. In general, the location that is used more frequently in a benchmark will be the most vulnerable unit for that particular benchmark. For example, if the benchmark is memory-intensive, the LSU will be one of the critical units for that benchmark. However, we find that some locations are vulnerable for all benchmarks. We report results for one integer benchmark (*astar*, Figure 10) and one FP benchmark (*dealII*, Figure 11). Other integer and FP benchmarks show similar behavior to the one described above.

In Figures 10 and 11, we only show the activated faults for the locations that result in program crashes. Locations that are not depicted (e.g., FP-to-Integer converter) did not lead to significant number of crashes because these locations generate values that are not critical in programs or they did not lead to activated faults due to very infrequent use.

Our data shows that the *Fetch-Inst* location results in relatively more crashes than other locations for both integer and FP benchmarks (82%, on average). Moreover, although FP benchmarks use the FPU unit heavily, most faults injected into the FP units result in benign faults. This is due to application masking which includes faults injected into bits that are not used by the application, especially the higher bits of a 64-bit registers and faults injected into registers used in evaluating logical operations. This result is consistent with the observation made by Li et al. [13], who reported 44% benign faults when permanent faults are injected into FP units.

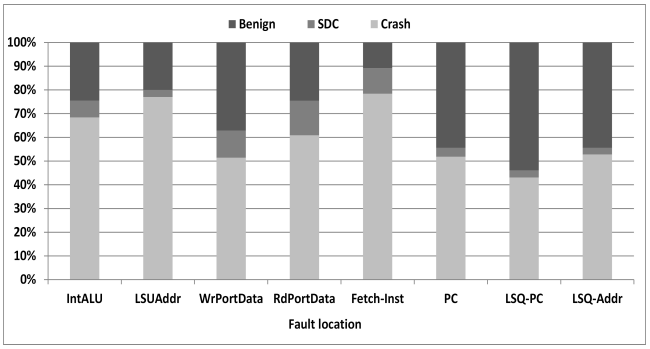

Fig. 10: Effect of intermittent fault location on how integer programs terminate (*astar* in this figure).

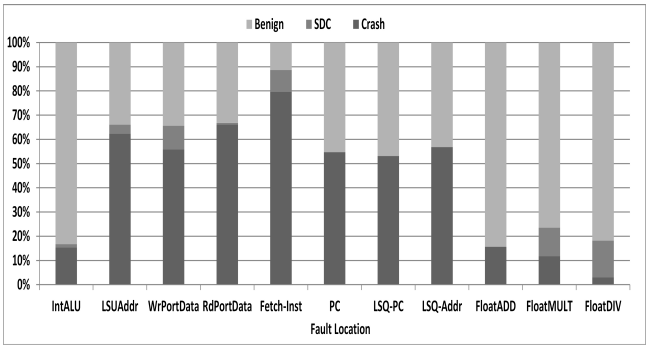

Fig. 11: Effect of intermittent fault location on how FP programs terminate (*dealII* in this figure).

Summary: We find that the percentage of crashes is sensitive to the intermittent fault length and location, but not significantly to the fault model used. Increasing the fault length causes an increase in the percentage of crashes until a certain threshold fault length, beyond which the percentage of crashes saturates. The impact of the fault model and the fault location is largely determined by how much corruption the corresponding model causes (stuck-at-one causes more corruption than stuck-at-zero, for example) and the criticality of the location at which the fault occurs (IntALU causes more crashes than LSQ-PC, for example).

VII Discussion

Our goal in this work is to study the usefulness of software-based techniques to diagnose and recover from intermittent hardware errors. While hardware techniques are transparent to the software, they have the following disadvantages:

1) Incur considerable performance or power overheads. For example, periodic testing techniques [7], [39], [40] can diagnose permanent and some types of intermittent faults caused by wearout. However, they incur high performance overhead even for fault-free cores, as the processor needs to be halted while the tests are executed (e.g., Constantinides et al. [40] report overheads of 30% for running online tests).
2) May initiate fault tolerance for faults that do not impact the application. For example, periodic testing and built-in-self-test [41] techniques examine all parts of the processor, even those parts that are rarely or never used by applications.
3) Typically do not support intermittent errors, which are non-deterministic in nature, and hence not easily reproducible.

Thus, there is a compelling need to develop intermittent fault tolerance techniques at the software level to complement existing hardware techniques. Software techniques have the following advantages: (1) work for non-deterministic faults, (2) do not require special hardware support, (3) do not run extra tests on the faulty processor (however, software-based techniques will require logging of the processor state, which incurs overhead), and (4) take into account the characteristics of the application, to only tolerate faults that affect the application. This is why we need software-level characterization of intermittent faults, as this paper has done.

We discuss below the impact of our characterization on software-based detection, diagnosis and recovery techniques. We also consider the impact of the sensitivity study.

Detection: Software-based detection techniques require careful placement of error detectors to prevent error propagation in programs. Our findings suggest that such techniques can be efficient because only 4% of intermittent errors propagate extensively (beyond one hundred thousand instructions) and hence require specialized detection techniques. However, since the IPS cardinality is less than 500 data values for 88% of the crash-causing errors, one should carefully place error detectors to cover the critical error propagation paths [42]. This requires analysis of the error propagation paths in the application. Software-anomaly based detection mechanisms that monitor hardware traps as indications of errors have been proposed for permanent errors [13]. Our results suggest that they would likely be effective for intermittent faults too, because the majority of such faults result in hardware traps or exceptions.

Diagnosis: Software diagnosis techniques aim to identify the fault-prone microarchitectural unit by analyzing the crash dump files caused by a fault [43]. For diagnosis to be efficient, it is important that the fault has limited propagation before causing programs to fail, and that there is sufficient information available in the crash dump file about the fault. Software-based diagnosis techniques can be efficient for intermittent faults because (1) the propagation sets for intermittent errors are limited to a few hundreds of dynamic instructions, (2) about 42% of the erroneous data is not masked, and about 46% of the erroneous registers are not overwritten by correct data. This means that many error "clues" that can be found on the program state are intact and can be used in isolating the defective microarchitectural part. However, more work is needed to learn more about how much of the microprocessor state is observable at the software-level and which parts of the processor can be diagnosed using such high-level software-based techniques.

Recovery: Checkpointing is a commonly used technique for recovering from transient faults. For checkpointing to be effective, it is important that errors cause programs to crash quickly, as otherwise, the error may propagate to a checkpoint and corrupt it. This would render the recovery process ineffective. Our results show that checkpointing techniques that gather checkpoints on the order of a few hundreds of thousands of instructions will be effective in recovering from intermittent errors. This is because the crash distances of such errors are less than a few hundreds of thousands of instructions and hence the error is unlikely to corrupt a checkpoint. Unlike transient faults, intermittent faults are likely to recur at the same microarchitectural location, and hence we need to reconfigure the processor around the faulty microarchitectural unit after restoring the program to the last stored checkpoint. We have shown in our previous study [28] that diagnosing and disabling the fault-prone microarchitectural component will result in higher performance than disabling the entire fault-prone core. Therefore, it is important to accurately diagnose the intermittent error after its occurrence and couple the checkpointing technique with fine-grained reconfiguration around the microarchitectural unit that caused the fault.

Sensitivity: In the sensitivity study, we found that both the intermittent fault length and location affect the percentage of faults that result in crashes. In particular, the processor's front-end is the most vulnerable part of the processor for both integer and FP benchmarks. Therefore, the processor designer can reduce the number of crashes that are caused by intermittent faults by hardening the front-end components (using larger transistors, for example). Further, errors in the LSU/LSQ affect the memory addresses used to read/write data, and hence this unit is vulnerable to intermittent faults and should be protected as well.

Moreover, we found that short faults lead to fewer crashes than longer ones. However, short faults may become longer if the extreme operating condition persist. Therefore, robust error detection techniques for short faults are necessary to avoid data corruption that would happen when the error becomes longer. On the other hand, if the processor is used for non-critical tasks and the application can tolerate limited data corruption, a short intermittent fault can be ignored (when a crash occurs, the program restored is to the last checkpoint)

until the error progresses to permanent one.

VIII Conclusions and Future Work

In this paper, we evaluated the impact of intermittent-hardware faults on programs through fault-injection experiments by monitoring how the injected programs terminate and measuring the crash distance and the error propagation for the failure (i.e., crash) causing errors. These factors are important because the further the point of failure is from the error origin and the more the error propagates, the more difficult it is to tolerate such an error. We find that the majority of the non-benign intermittent faults cause programs to crash. Further, the crash occurs within a hundred thousand dynamic instructions of the fault start for 96% of the crash-causing faults; hence large crash distances are infrequent. Finally, the number of dynamic data values corrupted by intermittent faults is less than 500 data values for about 88% of the crash-causing faults.

With the goal of examining the feasibility of software-based fault tolerance mechanisms, we study the usefulness of program's register file and memory footprint at crash-time in diagnosing intermittent errors. We accomplish this by measuring two parameters for each crash-causing error: number of erroneous data values masked by instructions and number of erroneous registers overwritten with correct values prior to crash. This is important because the more erroneous data values (or registers) that are masked (or overwritten) before the crash, the smaller the number of "clues" available about the error, and hence the higher the inaccuracy of software-based diagnosis techniques. Our results suggest that software-based diagnosis and recovery techniques can be efficient for intermittent faults.

As future work, we plan to build and evaluate software-based techniques for diagnosing and recovering from intermittent hardware faults. We also plan to extend our fault-injection mechanism to model other kinds of intermittent faults such as those caused by design errors. Finally, we will investigate the design of hardware-based mechanisms to support application-specific fault-tolerance for intermittent faults.

Acknowledgments

This research was supported by the National Science and Engineering Research Council (NSERC) through the Discovery Grant Program (Pattabirman, Gopalakrishnan) and by the Alexander Graham Bell Canada Graduate Scholarship (Rashid). We thank the anonymous reviewers of the IEEE Transactions on Reliability for their help in improving this paper.

References

[1] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, and A. Keshavarzi, "Parameter variations and impact on circuits and microarchitecture," *Design Automation Conf.*, 2003, pp. 338–342.

[2] C. Constantinescu, "Intermittent faults and effects on reliability of integrated circuits," *Reliability and Maintainability Symp.*, 2008, pp. 370–374.

[3] P. Wells, K. Chakraborty, and G. Sohi, "Adapting to intermittent faults in multicore systems," *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2008, pp. 255–264.

[4] E. Nightingale, J. Douceur, and V. Orgovan, "Cycles, cells and platters: An empirical analysis of hardware failures on a million consumer pcs," *European Conf. on Computer Systems*, 2011, pp. 343–356.

[5] J. McPherson, "Reliability challenges for 45nm and beyond," *Design Automation Conf.*, 2006, pp. 176–181.

[6] K. Bowman, S. Dunvall, and J. Meindl, "Impact of die-to-die and within-die parameter fluctuations on the maximum clock frequency distribution for gigascale integration," *IEEE Journal of Solid-State Circuits*, vol. 37, no. 2, 2002, pp. 183–190.

[7] J. Smolens, B. Gold, J. Hoe, B. Falsafi, and K. Mai, "Detecting emerging wearout faults," *Workshop On Silicon Errors in Logic-System Effects*, 2007.

[8] D. Ernst, N. S. Kim, S. Das, S. Pant, T. Pham, R. Rao, C. Ziesler, D. Blaauw, T. Austin, T. Mudge, and K. Flautner, "Razor: A low-power pipeline based on circuit-level timing speculation," *Intl. Symp. on Microarchitecture*, 2003, pp. 7–18.

[9] K. Skadron, M. Stan, K. Sankaranarayanan, W. Huang, S. Velusamy, and D. Tarjan, "Temperature-aware microarchitecture: Modeling and implementation," *ACM Transactions on Architecture and Code Optimization*, vol. 1, no. 1, 2004, pp. 94–125.

[10] W. Gu, Z. Kalbarczyk, R. Iyer, and Z. Yang, "Characterization of linux kernel behavior under errors," *Intl. Conf. on Dependable Systems and Networks*, 2003, pp. 459–468.

[11] J. Smolens, B. Gold, J. Kim, B. Falsafi, J. Hoe, and A. Nowatzyk, "Fingerprinting: Bounding soft-error detection latency and bandwidth," *Symp. on Architectural Support for Programming Languages and Operating Systems*, 2004, pp. 224–234.

[12] S. Kim and A. Somani, "Soft error sensitivity characterization for microprocessor dependability enhancement strategy," *Intl. Conf. on Dependable Systems and Networks*, 2002, pp. 416 – 425.

[13] M. Li, P. Ramchandran, S. Sahoo, S. Adve, V. Adve, and Y. Zhou, "Understanding the propagation of hard errors to software and implications for resilient system design," *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2008, pp. 265–276.

[14] N. Karimi, M. Maniatakos, A. Jas, and Y. Makris, "On the correlation between controller faults and instruction-level errors in modern microprocessors," *Intl. Test Conf.*, 2008, pp. 1–10.

[15] S. Pan, Y. Hu, and X. Li, "Ivf: Characterizing the vulnerability of microprocessor structures to intermittent faults," *Conf. on Design, Automation and Test in Europe*, 2010, pp. 238–244.

[16] S. Mukherjee, C. Weaver, J. Emer, S. Reinhardt, and T. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," *Intl Symp. on Microarchitecture*, 2003, pp. 29–41.

[17] N. Wang, A. Mahesri, and S. J. Patel, "Examining ace analysis reliability estimates using fault-injection," *Int. Symp. on Computer Architecture*, 2007, p. 460469.

[18] D. Gil-Tomas, J. Gracia-Moran, J.Baraza-Calvo, L. Saiz-Adalid, and P. Gil-Vicente, "Analyzing the impact of intermittent faults on microprocessors applying fault injection," *IEEE Design and Test of Computers*, vol. 29, 2012, pp. 66–73.

[19] L. Rashid, K. Pattabiraman, and S. Gopalakrishnan, "Formal diagnosis of hardware transient errors in programs," *SELSE Workshop-Silicon Errors in Logic-System Effects*, 2010.

[20] J. Wei, L. Rashid, K. Pattabiraman, and S. Gopalakrishnan, "Comparing the effects of intermittent and transient hardware faults on programs," *Workshop on Dependable and Secure Nanocomputing held in conjunction with the Intl. Conf. on Dependable Systems and Networks*, 2011, pp. 53 – 58.

[21] B. Huang, X. Li, M. Li, J. Bernstein, and C. Smidts, "Study of the impact of hardware fault on software reliability," *Intl Symp. on Software Reliability Engineering*, 2005, pp. 63–72.

[22] C. R. Elks, "Development of a fault injection-based dependability assessment methodology for digital and i&c systems," *United States Nuclear Regulatory Commission, Office of Nuclear Regulatory Research(NUREG/CR-7151)*, 2012.

[23] A. Ismaeel and R. Bhatnagar, "Test for detection and location of intermittent faults in combinational circuits," *IEEE transactions on reliability*, vol. 46, no. 2, 1997, pp. 269–274.

[24] F. Bower, D. Sorin, and S. Ozev, "Online diagnosis of hard faults in microprocessors," *ACM Transactions on Architecture and Code Optimization*, vol. 4.
[25] S. Park, T. Hong, and S. Mitra, "Ifra: Instruction footprint recording and analysis for post-silicon bug localization in processors," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, 2009, pp. 1545–1558.
[26] M. Li, P. Ramachandran, S. Sahoo, S. Adve, V. Adve, and Y. Zhou, "Trace-based microarchitecture-level diagnosis of permanent hardware faults," *Intl. Conf. on Dependable Systems and Networks*, 2008, pp. 22–31.
[27] A. Pellegrini and V. Bertacco, "Application-aware diagnosis of runtime hardware faults," *Intl. Conf. on Computer-Aided Design*, 2010, pp. 487–492.
[28] L.Rashid, K.Pattabiraman, and S.Gopalakrishnan, "Intermittent hardware errors recovery: Modeling and evaluation," *International Conference on Quantitative Evaluation of SysTems*, 2012, pp. 220–229.
[29] A. Avizienis, J. Laprie, and B. Randell, "Dependability and its threats: A taxonomy," *Int. Federation for Information Processing*, vol. 156, 2004, pp. 91–120.
[30] C. Weaver and T. Austin, "A fault tolerant approach to microprocessor design," *Int. Conf. on Dependable Systems and Networks*, 2001, pp. 411–420.
[31] M. Depas, M. Heyns, and P. Mertens, "Soft breakdown of ultra-thin gate oxide layers," *European Solid State Device Research Conf.*, vol. 25, 1995, pp. 235–238.
[32] N. Wang, A. Mahesri, and S. J. Patel, "Material dependence of hydrogen diffusion: implications for nbti degradation," *Int. Electron Devices Meeting. IEDM Technical Digest*, 2005, p. 691695.
[33] W. Tisdale, K. Williams, B. Timp, D. Norris, E. Aydil, and X. Zhu, "Hot-electron transfer from semiconductor nanocrystals," *Science Magazine*, vol. 328, no. 5985, 2010, pp. 1543–1547.
[34] J. Abella and X. Vera, "Electromigration for microarchitects," *ACM Computer Survey*, vol. 42, no. 2, 2010, pp. 9:1–9:18.
[35] K. Kim, R. Jayabharathi, and C. Carstens, "Speedgrade: an rtl path delay fault simulator," *Asian Test Symp*, 2001, pp. 239 – 243.
[36] D. Burger and T. Austin, "The SimpleScalar tool set, version 2.0," *Computer Architecture News*, vol. 25, no. 3, 1997, pp. 13–25.
[37] H. Najaf-Abadi, "Simplescalar ported to alpha/linux," *http://hhnajafabadi.s3-website-us-east-1.amazonaws.com/mase-alphalinux.htm*.
[38] R. Joseph, D. Brooks, and M. Martonosi, "Control techniques to eliminate voltage emergencies in high performance processors," *Intl. Symp. on High-Performance Computer Architecture*, 2003, pp. 79–90.
[39] Y. Li, S. Makar, and S. Mitra, "Casp: Concurrent autonomous chip self-test using stored test patterns," *Conf. on Design and automation and test in Europe*, 2008, pp. 885–890.
[40] K. Constantinides, O. Mutlu, T. Austin, and V. Bertacco, "Software-based online detection of hardware defects: Mechanisms and architectural support and evaluation," *Intl. Symp. on Microarchitecture*, 2007, pp. 97–108.
[41] S. Shyam, K. Constantinides, S. Phadke, V. Bertacco, and T. Austin, "Ultra low-cost defect protection for microprocessor piplines," *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, vol. 41, no. 11, 2006, pp. 73–82.
[42] K. Pattabiraman, Z. Kalbarczyk, and R. Iyer, "Application-based metrics for strategic placement of detectors," *Pacific Rim Intl. Symp. on Dependable Computing*, 2005, pp. 75–82.
[43] L. Rashid, K. Pattabiraman, and S. Gopalakrishnan, "Dieba: Diagnosing intermittent errors by backtracing application failures," *Silicon Errors in Logic - System Effects*, 2012.

Layali Rashid received her Ph.D. from the University of British Columbia in 2013 and her M.S. from the University of Calgary in 2007. She is a Senior Engineer in the CPU Performance Modeling team at Qualcomm Technologies. Her research interests include designing reliable and high performance processors. Layali was the recipient of Alexander Graham Bell Canada Graduate Scholarship.

Karthik Pattabiraman received his M.S and PhD. degrees from the University of Illinois at Urbana-Champaign (UIUC) in 2004 and 2009 respectively. After a post-doctoral stint at Microsoft Research (Redmond), Karthik joined the University of British Columbia (UBC) as an assistant professor of electrical and computer engineering. Karthik's research interests include programming languages, compilers and computer architecture for building error resilient software systems. Karthik has won a best paper award at the IEEE International Conference on Dependable Systems and Networks (DSN), 2008, a best paper runner up award at the IEEE International Conference on Software Testing (ICST), 2013 and a Distinguished paper award at the IEEE/ACM International Conference on Software Engineering (ICSE), 2014. Karthik was recently the general chair for the IEEE Pacific Rim International Symposium on Dependable Computing (PRDC), 2013.

Sathish Gopalakrishnan is an associate professor of Electrical and Computer Engineering at the University of British Columbia. His research interests center around resource allocation problems in several contexts including real-time, embedded systems and wireless networks. Prior to joining UBC in 2007, he obtained a Ph.D. in Computer Science and an M.S. in Applied Mathematics from the University of Illinois at Urbana-Champaign. He has received awards for his work from the IEEE Industrial Electronics Society (Best Paper in the IEEE Transactions on Industrial Informatics in 2008) and at the IEEE Real-Time Systems Symposium (in 2004).