Modeling the Propagation of Intermittent Hardware Faults in 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. Recent studies have shown that intermittent fault rates are increasing due to technology scaling and are likely to be a significant concern in future systems. We study the impact of intermittent hardware faults in programs. A simulation-based fault-injection campaign shows that the majority of the intermittent faults lead to program crashes. We build a crash model and a program model that represents the data dependencies in a fault-free execution of the program. We then use this model to glean information about when the program crashes and the extent of fault propagation. Empirical validation of our model using fault-injection experiment shows that it predicts almost all actual crash-causing intermittent faults, and in 93% of the considered faults the prediction is accurate within 100 instructions. Further, the model is found to be more than two orders of magnitude faster than equivalent fault-injection experiments performed with a microprocessor simulator.**

I. Introduction

Intermittent hardware faults are bursts of errors that occur at the same location (i.e., micro-architectural component) and last from a few cycles to a few seconds [1]. They can occur due to a variety of causes such as variations in device reliability [2], fluctuations in supply voltage and temperature (PVT) [3], [4], in-progress wear-outs and manufacturing residues [1], [5]. 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], [4], [6].

Fault avoidance techniques mitigate intermittent faults by minimizing process variations [4], regulating voltage [7] or managing temperature [8]. Although such techniques reduce the base rate of intermittent faults, many such faults still occur and escape to the software [6]. Therefore, we need mechanisms at the software level to mitigate the impact of intermittent faults through detection, diagnosis and recovery.

We study and model the impact of intermittent faults on software programs as a first step towards building software-level mechanisms for mitigating intermittent faults. To the best of our knowledge, this is the first study of intermittent fault propagation at the program level. Prior work has analyzed the effects of transient faults and permanent faults on software programs [9], [10]. However, intermittent faults are unlike transient faults in that they affect more than one instruction, and are unlike permanent faults in that they do not persist at the same location. 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].

A hardware fault can affect programs in different ways. The fault may be benign (it has no impact on the program), cause silent data corruption (i.e., incorrect program output) or lead to a program crash. Even if a fault does lead to a crash, the crash may not occur immediately after the onset of the fault, but only after some amount of time later (crash latency). In the meantime, the fault may propagate to other instructions in the program and corrupt their data values. 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 do they propagate within their programs before the crashes?
We focus on crashes, in this study, because our experiments indicate that more than 95% of non-benign intermittent faults result in program crashes (Section IV-A).

The questions asked above can be answered by subjecting benchmark programs to a fault-injection campaign using a micro-architectural simulator. However, this method is prohibitively expensive because fault-injection experiments involve a significant time investment, especially when we want to inject all possible intermittent faults, which vary in both location and duration. Further, fault injections yield limited insight for characterizing the propagation of intermittent faults at the program level. Therefore, we need efficient models of fault propagation at the program level for designing error mitigation mechanisms in software.

We consider the applicability of Dynamic Dependency Graph (DDG) [11], [12], an instruction-level model for representing program dependencies, in modeling and understanding the propagation of intermittent faults. A DDG is derived from a dynamic instruction trace of a program obtained from a fault-free execution of that program. Thus, unlike a fault-injection campaign, we execute a program only once and gather its instruction trace to construct its DDG. This allows further analysis to be performed more than two orders of magnitude faster than an equivalent fault-injection campaign.

The contributions of our work are as follows: (1) Characterizing the impact of intermittent faults on programs through fault-injection experiments carried out using the SimpleScalar simulator [13]. (2) Modeling the propagation of intermittent faults in programs at the instruction level using the DDG. (3) Validating the DDG as a model for examining fault propagation. This is accomplished by comparing the model's predictions with equivalent intermittent-fault injection experiments on seven benchmark programs [14].

The main results of the study are as follows:

- The predominant impact of an intermittent fault on software is a program crash. Between 62% and 79% (varied by benchmark) of the faults we injected led to program crashes. Of the remaining fault injections, between 20% and 34% were benign, and only about 4% of injected faults resulted in silent data corruption.
- Using the DDG model, we were able to predict almost all the crashes observed in the fault-injection experiments. The false negative rate of the predictions was at most 0.06% and the false positive rate was about 20%.
- The DDG model's predictions of a fault's crash location and propagation were accurate to within 100 dynamic instructions of the actual values for about 93% of the faults. The discrepancies in the remaining faults were due to aspects of program execution that were not captured by the DDG model, such as errors affecting stack pointers.
- Analysis using the DDG model required 4 to 89 seconds per benchmark, while the equivalent fault injections with the SimpleScalar simulator required 6 to 70 hours per benchmark.

II. METHOD

In this section, we: (1) describe our fault model, and create a crash model to reason about where programs crash due to intermittent faults, (2) define two metrics to characterize the propagation of intermittent faults in programs, and (3) show how to calculate the values of those metrics using a Dynamic Dependency Graph (DDG) [11] model of the program.

A. Fault Model

While intermittent faults may manifest themselves in programs in many ways, we focus on fault bursts that affect multiple consecutive instructions in an execution of a program. In our model, an intermittent fault can be specified by the program instruction that is being executed when the fault starts and by the number of instructions over which the fault persists (the length). We also assume that all instructions within a fault burst are affected by the fault. Our fault model can capture different fault types including:

- Instruction decoding errors that result in a different instruction being executed than the intended one.
- 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.
- Load/store unit errors that result in loading/storing data from/to incorrect memory locations.

We assume that the processor's memory hierarchy (caches and main memory) is reliable and hence do not consider memory/cache errors. This is reasonable as such errors can be detected by parity or ECC (in memory). We also assume that the processor's control logic is error-free. This is also reasonable because the control logic constitutes a very small portion of the processor [15].

B. Crash Model

It is important to build an accurate model of when programs crash due to an error. This is because all errors, and more importantly the propagation of all errors within programs, will of course end at the instance of a program crash. We focus on crashes because they are the most common consequences of intermittent faults (as we show in Section IV). Moreover, when encountering a crash-causing error, programs do not necessarily crash immediately. They rather continue executing, leading to error propagation. This in turn can result in extended down-times or corruption of permanent state [9]. Hence, crashes are not always benign or fail-stop.

Previous studies (e.g., [16]) have shown that executing instructions that use erroneous pointer values will likely lead to program crashes at the same instruction. The crash model therefore assumes that a program crashes when a fault propagates to any of the following:

1) memory address in a load or store instruction,
2) destination address of a branch instruction or
3) destination or return addresses of a function call.

This crash model is conservative and over-approximates the set of circumstances that may lead to a crash. We study the consequences of this over-approximation in Section IV-D.

C. Terms and Definitions

In the definitions below, a transient fault is assumed to affect a single instruction, while intermittent faults affect two or more instructions in the program.

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.

Dynamic Dependency Graph (DDG) is a directed acyclic graph that models the dynamic data dependencies among nodes generated during a program's execution. The dependencies are represented as edges in the graph, and are annotated based on the type of dependency (address/regular). DDGs have been used in prior work for understanding and analyzing program behavior [11], [12], [17]. To the best of our knowledge, we are the first to use DDGs in modeling the propagation of intermittent errors.

$CrashNode(i)$ is the node at which a program crashes due to a transient fault at node i (as per the crash model).

Transient Propagation Set or $TPS(i)$ is the set of nodes to which a transient fault that occurs at node i propagates until it reaches $CrashNode(i)$.

$CrashNode(i_s, i_e)$ is the node at which a program crashes due to an intermittent fault that starts at node i_s and ends at another node i_e, where i_s and i_e are DDG nodes such that $i_s < i_e$ (as per the crash model).

Intermittent Propagation Set or $IPS(i_s, i_e)$ is the set of nodes to which an intermittent fault, that starts at node i_s and ends at i_e propagates until it reaches $CrashNode(i_s, i_e)$ or i_e, whichever comes later.

Crash Distance or $CD(i_s, i_e)$ is the number of nodes that

are generated by a program from the start of an intermittent fault occurring between node i_s and node i_e and until $CrashNode(i_s, i_e)$ is reached.

D. Computing the Intermittent Fault Propagation Set

In order to compute the propagation set for an intermittent fault in a program (IPS), a DDG is constructed from a trace file of a fault-free program execution. The construction of the DDG uses established techniques [11], [17] and hence will not be elaborated here. Because we use $TPS(i)$ in computing $IPS(i_s, i_e)$, we first explain how to compute $TPS(i)$ below. The following steps are repeated for every node i in the DDG model (in topological sort order of nodes[1]).

1) Compute $CrashNode(i)$ by recursively following the data paths of all successors of the node i in best-first fashion, which relies on the order in which nodes appear in a fault-free run, and stopping when a potential crash point is reached (as per the crash model). Note that while a node may have multiple crash nodes, we only need the earliest crash node since the execution stops thereafter.
2) Compute $TPS(i)$, by performing a best-first search starting from the successors of node i until we reach $CrashNode(i)$. The TPS consists of all nodes visited in the best-first search traversal. We consider every node in the DDG model as a potential starting point for an intermittent fault, and consider all possible intermittent fault lengths from that node. For each intermittent fault (i_s, i_e), we do the following:
3) Compute $CrashNode(i_s, i_e)$ as the earliest crash node among all crash nodes of the nodes between i_s and i_e (inclusive), i.e., it is the minimum of $CrashNode(i)$, $i \geq i_s$ and $i \leq i_e$.
4) Compute $CrashDistance(i_s, i_e)$ as the number of nodes between the start of the intermittent fault, namely i_s and the crash node of the intermittent fault, namely $CrashNode(i_s, i_e)$.
5) Compute $IPS(i_s, i_e)$ as the union of the $TPS'(i)$ such that i is between i_s and i_e (inclusive), where $TPS'(i)$ is equal to $TPS(i)$ computed with $CrashNode(i)$ equal to $CrashNode(i_s, i_e)$. In other words, an IPS includes all nodes in the TPSs of nodes over which the intermittent fault spans, until the $CrashNode(i_s, i_e)$ is reached, after which no more nodes are added.

E. Example of IPS/CD Computation

In this section, we demonstrate how to compute the CD and IPS of an intermittent fault using an example code fragment from Table I. 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 general, loops do not present a problem as the DDG model is constructed from a dynamic execution of a program and hence all loop iterations are unrolled in execution. In the example code fragment, elements at indices 5, 6 and 7 of an array (starting

[1]Recall that a DDG is a directed acyclic graph and hence it has no cycles.

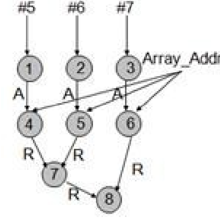

Fig. 1. The DDG corresponding to code fragment in Table I.

TABLE I
CODE FRAGMENT TO ILLUSTRATE IPS/CD COMPUTATION.

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

at $Array_Addr$) 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.

Figure 1 shows the DDG for the code fragment in Table I. The last column of the table shows mapping from instructions to DDG nodes. In the figure, nodes are drawn as circles with the node number inside the circle. The edges represent the type of dependencies among nodes and are labeled with the operand's type, where `A` is an address operand and `R` is a register operand. Assume that an intermittent fault affects the first two instructions in Figure 1. We want to compute $IPS(1, 2)$. Because we use $TPS(i)$, $i = \{1, 2\}$ to compute $IPS(1, 2)$, we first explain how to compute $TPS(i)$. Recall that $TPS(1)$ is the set of nodes to which a transient error at node 1 propagates. Hence, it includes at least the node at which the error occurs, namely node 1. Node 1 has a successor node 4, which corresponds to the dynamic instance of the instruction $ld\ R4,\ R1,\ Array_Addr$. Hence $TPS(1) = \{1, 4\}$. However, node 1 is used as an address operand in the instruction (`A` operand), hence the program crashes at this instruction (as per the crash model). Therefore, $CrashNode(1) = 4$ and $TPS(1) = \{1, 4\}$. Similarly, we can compute $TPS(2) = \{2, 5\}$ and $CrashNode(2) = 5$.

To compute $IPS(1, 2)$, we first find $CrashNode(1, 2)$, which is the minimum of the crash nodes for nodes between 1 and 2 (inclusive). $CrashNode(1, 2) = min(CrashNode(1), CrashNode(2)) = CrashNode(1) = 4$. We then compute the union of $TPS(1)$ and $TPS(2)$, only including nodes at or before $CrashNode(1)$. Consequently, $IPS(1, 2) = \{1, 2, 4\}$. Note that node 5 is not included in $IPS(1, 2)$ even though it was included in $TPS(2)$ because it is generated after its crash node 4. $CrashDistance(1, 2)$ is the number of nodes generated from the start of the intermittent fault until its crash node 4, i.e.,

TABLE II
BENCHMARKS AND THEIR DESCRIPTIONS.

Benchmark	Description
Tot info	Offers a series of data analysis functions
Print Tokens	Breaks the input stream into a series of lexical tokens according to pre-specified rules
Replace	Searches a text file for a regular expression and replaces the expression with a string
Tcas	Aircraft collision avoidance application
Print Tokens2	Breaks the input stream into a series of lexical tokens according to pre-specified rules
Schedule	A priority scheduler for multiple job tasks
Schedule2	A priority scheduler for multiple job tasks

$CrashDistance(1,2) = |\{1,2,3,4\}| = 4.$

III. EXPERIMENTAL SETUP

We now describe the experimental setup used to study the impact of intermittent faults on programs and to validate the DDG model. We used the Siemens benchmark suite [14], which comprises seven programs written using the C language. The programs range between 100 to 1000 lines of code (Table II). For each program, we chose arbitrary input from the corresponding test suite. We conducted our experiments on an Intel Xeon E5345 Quad Core 2.33GHz system with 8GB main memory and 4MB of L2 cache. The experiments consisted of two phases: the first phase used the DDG model to compute IPS and CD, while the second phase evaluated the accuracy of the DDG model using fault-injection experiments.

Phase 1: Computing the propagation set and the crash distance

We modified sim-safe from the SimpleScalar simulator [13] to capture the instruction trace of a fault-free program execution. For each instruction executed by the program, SimpleScalar recorded the instruction's type and the node(s) used and generated by the instruction to a trace file. It also recorded the order in which the dynamic instructions were executed (i.e., the program's control flow). Based on this instruction trace, we constructed the DDG model of the program, and considered potential intermittent faults. These faults ranged in length from 2 to 10 instructions and would start at any node in the DDG, one fault at a program run (the maximum fault length we considered was 10 nodes since we found that the average CD for 90% of intermittent faults was 8 nodes). If the fault resulted in a crash, we calculated its expected IPS and CD values using the DDG model. If it did not result in a crash, we ignored the fault since we focused only on crash-causing faults in this study.

Phase 2: Evaluating the accuracy of the DDG model

We injected the faults considered by the DDG into a modified SimpleScalar simulator [13]. We modified sim-safe from SimpleScalar to mimic the behavior of the operating system when an application crashes [17]. As before, the fault length ranged from 2 to 10 nodes. For each one of these intermittent faults, SimpleScalar executed the application and injected a single fault (a single bit-flip into a randomly chosen byte) into the corresponding output registers of the instructions that are spanned by the fault. If the program crashed, SimpleScalar generated a crash dump file that contained the node and the instruction PC at which the program crashed and a list of erroneous nodes in the program (i.e., the IPS of the fault)[2]. Otherwise, it executed the application to completion and compared the application's output with the expected output (golden run). In case of a mismatch, the fault is considered a Silent-Data Corruption (SDC) fault. Otherwise it is considered a benign fault, as it has no effect on the program.

We counted the number of faults that were predicted to cause crashes but did not actually do so in SimpleScalar (false positives), and the faults that caused crashes in SimpleScalar but were not predicted to do so using the DDG model (false negatives). For the faults that were predicted to cause crashes and actually did so in SimpleScalar, we compared their CD and IPS values with the values predicted using DDG (for the same faults).

IV. RESULTS

In this section, we first characterize the impact of intermittent faults on programs by injecting faults using SimpleScalar [13]. We measure the percentage of crashes, silent data corruptions and benign faults for the injected faults (Section IV-A). We then assess the accuracy of the DDG model along three different dimensions: (1) number of crashes, (2) crash distances, and (3) propagation sets. Further, we focus on the differences between the expected and actual crash distances to understand the reasons behind these variations (Section IV-D). Finally, we examine the relationship between the fault length(s) and the predicted and real CDs and IPSs (Section IV-E).

Performance: We measure the time taken by DDG and SimpleScalar-based fault injections to obtain the results reported in this section. Table III shows the execution times (in seconds) for each program. In addition, we measure the total number of DDG nodes generated by each benchmark and the total number of intermittent faults considered for the program[3]. The DDG model takes 4 to 89 seconds to find all potential crash-causing faults. In comparison, the fault-injection experiments in SimpleScalar require about 21600 to 252000 seconds (6 to 70 hours) to complete for the same set of intermittent faults.

To further study how time is spent in the DDG model, we present the breakdown of the DDG execution time for the largest benchmark, schedule2. For this program: the trace-gathering run takes 3.0 seconds, DDG construction takes 18.5 seconds and computation of the metrics takes 67.1 seconds. Note that the DDG-construction and trace-gathering overheads are both one-time costs that can be amortized over the number of intermittent faults considered. Moreover, the time complexity of building the DDG is linear with respect to the number

[2]Our modified version of the simulator maintained DDG information even during the fault-injections phase. We assumed that nodes corresponded to the same PCs in DDG and SimpleScalar.

[3]Number of faults injected into a benchmark depends on the number of instructions in that benchmark.

TABLE III
REQUIREMENTS OF THE DDG MODEL AND SIMPLESCALAR INJECTIONS IN NUMBER OF NODES AND TIME IN SECONDS.

Benchmark	No. of Nodes	No. of Faults	Time Req. by the DDG	Time Req. by the SS
Tot_info	25640	230715	11	43200
Print_Tokens	26366	237276	11	46800
Replace	12417	111735	6	28800
Tcas	8532	76770	4	21600
Print_Tokens2	24081	216711	11	39600
Schedule	137777	1239948	53	244800
Schedule2	210875	1897830	89	252000

TABLE IV
BREAKDOWN OF FAILURE CATEGORIES IN SIMPLESCALAR(SS).

Benchmark	Rate of SS Crashes	Rate of SDC	Rate of Benign Faults
Tot info	73.94%	5.11%	20.95%
Print Tokens	62.07%	4.01%	33.91%
Replace	72.76%	4.10%	23.13%
Tcas	73.62%	3.01%	23.38%
Print Tokens2	64.95%	3.65%	31.39%
Schedule	79.02%	3.04%	17.94%
Schedule2	75.56%	2.26%	22.18%

of instructions.

Thus, *DDG is more than two orders of magnitude faster than the SimpleScalar-based fault injections for the applications studied, and is a more scalable technique for evaluating the effects of intermittent faults on programs.* Future work will examine the scalability of the DDG to even longer programs.

A. Classifying Intermittent Faults Impact

In this subsection, we study the effects of intermittent faults on programs using SimpleScalar-based fault injections. The experiments in this section do not involve the DDG (Table IV).

Of the total number of faults injected, we find that 70.52% result in crashes, 3.43% in Silent-Data-Corruption (SDC), and the remaining 26.05% in benign outcomes, i.e., they do not alter the program's output. Thus, of the intermittent faults that are non-benign (i.e., 73.95% of total faults), 95.11% result in a program crash. This high percentage reaffirms our focus on crash-causing intermittent faults in this study.

Moreover, we find that 34.39% of the total crashes (not shown in Table IV) are truncated, i.e., the intermittent fault causes a crash in the very first affected instruction. Most of these truncated intermittent faults affect memory-instructions such as loads, stores and jump instructions.

B. Intermittent Faults' Crash Distances

We present the absolute values of the crash distances for the faults injected in SimpleScalar. A significant number of the crash distances (99.71% on average) are within 10000 instructions. However, there is a small fraction of intermittent faults (0.29%) that have larger CDs than 10000 (Table V). We find that at-most 0.28% of faults result in crashes beyond one hundred thousand instructions from their start nodes. Such faults result either in changes to the control flow of the program or in very deep loop iterations.

TABLE V
CRASH DISTANCES FOR LONG LATENCY INTERMITTENT FAULTS IN SIMPLESCALAR.

Benchmark	$\leq$ 10000	10000-100000	100000-1000000
Tot info	99.52%	0.20%	0.28%
Print Tokens	99.81%	0.14%	0.05%
Replace	99.88%	0.09%	0.03%
Tcas	99.69%	0.28%	0.03%
Print Tokens2	99.82%	0.13%	0.05%
Schedule	99.62%	0.27%	0.11%
Schedule2	99.80%	0.09%	0.11%

C. Evaluating the Accuracy of the Expected Number of Crashes

We assess the accuracy of the DDG model by comparing its predictions to the fault-injection results obtained from the SimpleScalar-based fault-injection campaign (Section III). These comparisons include: (1) the difference between intermittent faults that are expected to lead to crashes and those that actually do so (Table VI), (2) the difference between the predicted intermittent faults' CDs and the actual ones (Figure 2), and (3) the differences between the predicted intermittent faults' IPSs and the actual ones (Figure 3).

We measure the percentage of intermittent faults that are expected to result in crashes and actually do so in SimpleScalar (i.e., true expected crashes). We also measure the percentage of faults causing crashes that occur in SimpleScalar but are not predicted by the DDG model (i.e., false negatives). Table VI shows the results.

Table VI shows that 74% to 81% of the faults in the total expected crashes category are true expected crashes, i.e., they crash in both DDG and SimpleScalar. The remaining faults do not lead to a crash, but rather lead to benign outcomes and SDCs in SimpleScalar. Hence, they are false-positives. This inaccuracy arises because the crash model used in the DDG (Section II-B) is conservative and hence over-estimates the number of crashes that occur in reality (i.e., in SimpleScalar). Further, Table VI also shows that 0.00% to 0.06% of the total number of crashes in SimpleScalar are not predicted by DDG. These are false negatives for the DDG model. We find that these missed crashes are due to faults that are either injected into arithmetic instructions that update the stack pointer or into registers that store very large values (hence, injecting a fault into these registers would likely cause an integer overflow, which is not modeled by the DDG).

Moreover, most of the mis-predicted intermittent faults in each benchmark start at the same node (a node is a value produced by a dynamic instruction) which suggests that they are localized to specific code segments. Future work will attempt to improve the DDG model to account for these faults.

To quantify the accuracy of the predicted crash distances (i.e., the total number of nodes generated by a program from the start of an intermittent fault to the point of the crash),

TABLE VI
THE PERCENTAGES OF TRUE EXPECTED CRASHES AND MISSED CRASHES FOR EACH BENCHMARK.

Benchmark	Percentage of True Expected Crashes	Percentage of Missed Crashes
Tot Info	74%	0.01%
Print Tokens	78%	0.03%
Replace	79%	0.00%
Tcas	77%	0.00%
Print Tokens2	76%	0.06%
Schedule	81%	0.03%
Schedule2	78%	0.05%

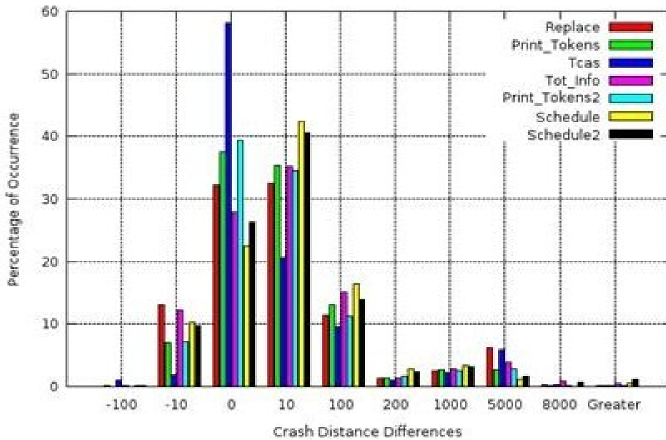

Fig. 2. The difference between actual CDs in SimpleScalar and the corresponding CDs as predicted by the DDG model.

we compare the Crash Distances (CDs) of the injected faults in SimpleScalar to those predicted by the DDG model. We measure the percentages of faults whose CDs differ from their predicted CDs (independent of the fault's start node and its length) and plot their distribution with respect to the differences in Figure 2.

We find that for 86% of the faults whose CDs differ, the DDG model under-estimates the CD (i.e., the actual CD is larger than the corresponding predicted CD). This is because DDG is conservative in predicting crashes. However, we find that 75 to 82% of the predicted crash distances are within 10 nodes and, 89 to 93% are within 100 nodes of the actual crash distances for all programs. This shows that the DDG model is accurate in predicting the CDs of crash-causing faults to within 100 nodes of their actual CDs.

To evaluate the accuracy of the Intermittent Propagation Set (i.e., the number of nodes affected by the intermittent fault before the program crashes, IPS) by the DDG model, we compare the IPSs resulting from the injected faults in SimpleScalar to those predicted by the DDG model. We compute the difference set of the two sets and measure the cardinality of this set (i.e., the number of nodes that are different). Figure 3 shows the percentage distribution of the difference set's cardinalities.

We find that 88 to 94% of the difference sets' cardinalities are within 10 nodes and that 93 to 97% are within 100 nodes

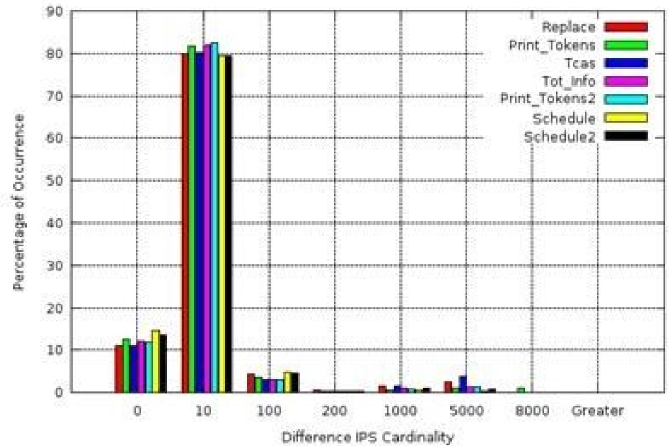

Fig. 3. The cardinality of the difference set of actual IPSs in SimpleScalar and the corresponding IPSs as predicted by the DDG model.

across all programs. Thus, the DDG model is highly accurate in predicting the IPS of intermittent faults that cause crashes to within 100 nodes of their actual IPS.

Further, the divergence between the expected and actual IPS is even smaller than the observed divergences for the CDs. Hence, out of the faults whose CDs deviate by more than 100 nodes from the expected CDs, only about half exhibit deviations in the IPS by more than 100 nodes. This illustrates that faults that take a long time to crash do not always propagate extensively in the program.

In summary, the DDG is highly accurate in predicting both CDs and IPSs of crash-causing faults in SimpleScalar (to within 100 nodes). However, there are also large variations between the predicted and observed values in 5 to 10% of the faults. We elaborate on the reasons for these variations in the next sub-section.

D. Understanding the Differences between DDG and SimpleScalar

To understand the rationale for the deviations between expected and actual intermittent faults properties (CD and IPS), we classify the faults predicted by DDG into two categories: faults with low CDs and faults with high CDs (Table VII), where low means less than or equal to 100 nodes and high means larger than 100 nodes. We further divide each category into three sub-categories based on their manifestation in SimpleScalar: low CDs, high CDs or no crashes. We perform a similar categorization based on the IPS cardinalities (Table VIII). We show results for the tot_info benchmark. Similar results are obtained for the other programs and are hence not reported.

The following results may be observed from Table VII and Table VIII:

- 99.98% of crashes in SimpleScalar are predicted by the DDG model (Table VI). However, not all DDG-predicted crashes induce SimpleScalar crashes, but only 74% of them do so. The remaining faults are false positives for

TABLE VII
EVALUATING THE PREDICTED CRASH DISTANCES FOR TOT INFO BENCHMARK.

	Low Actual CD	High Actual CD	No Crashes
Low Exp. CD (99.98%)	66.40%	7.51%	26.07%
High Exp. CD (0.02%)	0.003%	0.00%	0.017%

TABLE VIII
EVALUATING THE PREDICTED IPS CARDINALITIES (IPSC) FOR TOT INFO BENCHMARK.

	Low Actual IPSC	High Actual IPSC	No Crashes
Low Exp. IPSC (100.00%)	71.67%	2.26%	26.07%
High Exp. IPSC (0.00%)	0.00%	0.00%	0.00%

the DDG model. Thus, DDG over-predicts the number of real crashes (as shown in Section IV-C).

- Of the 74% that the DDG model correctly predicts as crashes, it under-predicts the CD for about 9.85% (7.51% out of the 74%) of them. In other words, it predicts that all 74% of the faults crash fairly quickly (within 100 instructions), while in reality, only 90.15% of these faults do so.
- Of the 9.85% of the faults whose CDs are under-predicted, the DDG model assumes that all of them generate a low IPS cardinality (Table VIII). However, only 95.65% of the faults satisfy this criterion, whereas the remaining 4.35% of the faults have both high CDs and high IPS cardinalities.

Thus, the DDG model over-predicts the number of faults that cause crashes, but for the ones that it correctly predicts, it under-predicts the CD by about 9.85%. In these cases, it also under-predicts the IPS (for 4.35%). Thus, the DDG model accurately predicts CDs and IPSs for the majority of the faults that actually cause crashes with low crash distances. However, there are discrepancies for faults with high CDs. To gain a better understanding of the discrepancy, we demonstrate two examples where a fault results in a crash with a high CD and a high or a low IPS cardinality and the DDG predicts low CDs for the same faults.

Example of a fault with high CD and high IPS cardinality: Table IX shows an example code sequence taken from the library function memset: The code sequence consists of a loop to store data (a3[7]) to a set of locations with pre-calculated base-address ($v0[2]) and different indexes (0, -4, ..., -24), (PCs 0x40d608 to 0x40d638). It then updates the base-address at PC 0x40d640 and the loop counter v1[3] at PC 0x40d650, and executes another loop-iteration if the counter is not zero (PC 0x40d658).

Consider an intermittent fault of length 7 instructions that affects PCs 0x40d608 to 0x40d638, which correspond to nodes 2496to 2503 in the DDG. Based on the DDG's crash model, a crash is predicted at the first encountered store instructions (0x40d608 <memset+98> sw $a3[7],-24($v0[2])). Therefore, the predicted CD is equal to zero and the predicted IPS only contains the injected nodes (2496 to 2503) and hence has a cardinality of 7.

When the intermittent fault is injected into the application executing in SimpleScalar, a least-significant-bit (bit 7 in this example) is flipped in the address of the store instruction, and the fault in register $v[0] persists without causing a crash until a later loop iteration at node 5339 (0x40d608 <memset+98> sw $a3[7],-24($v0[2])). At this point, the program attempts to store the data in the erroneous address, upon which it crashes. Consequently, the real CD is 2843 (5339 to 2496), and the IPS cardinality is 1896, which represents all the values that are stored in the wrong addresses until the program crashes. Thus, the fault has a high CD and a high IPS cardinality.

Example of a fault with high CD and low IPS cardinality:

TABLE IX
EXAMPLE CODE FRAGEMENT WITH HIGH CD AND HIGH IPS CARDINALITY.

```
0x40d608 <memset+98> sw $a3[7],-24($v0[2])
0x40d610 <memset+a0> sw $a3[7],-20($v0[2])
0x40d618 <memset+a8> sw $a3[7],-16($v0[2])
0x40d620 <memset+b0> sw $a3[7],-12($v0[2])
0x40d628 <memset+b8> sw $a3[7],-8($v0[2])
0x40d630 <memset+c0> sw $a3[7],-4($v0[2])
0x40d638 <memset+c8> sw $a3[7],0($v0[2])
0x40d640 <memset+d0> addiu $v0[2],$v0[2],32
0x40d648 <memset+d8> addiu $t0[8],$t0[8],32
0x40d650 <memset+e0> addiu $v1[3],$v1[3],-1
0x40d658 <memset+e8> bne $v1[3],$zero[0],0x40d600
```

Table X shows another example code fragment from the malloc library function. When the code sequence is executed, register $v0[2] has a value greater than zero. An intermittent fault of length 6 affects instructions with PCs 0x403fc8 to 0x403f88 in the code, which corresponds to nodes 15775 to 15781 in the DDG. Since the first instruction in the intermittent fault interval is a branch, it is assumed to be the crash node by the DDG's crash model. Hence, the predicted CD is zero and the expected IPS consists of the injected nodes only (15775 to 15781).

In the real execution, the fault that is injected into the first instruction of the code sequence results in random bit flip (bit 7 in this example) and modifies the branch destination address from 0x403f60 to 0x403fe0. This effectively causes the branch to fall through, and the control to transfer to PC 0x403fe0. This instruction is also a branch instruction and is taken. In this case too, the control is transferred to the instruction at PC 0x403f70 instead of the original destination at PC 0x403ff0. Then the intermittent fault modifies the addresses of the data loaded or stored and the values computed at PCs 0x403f70, 0x403f78, 0x403f80 and 0x403f88 (the fault length is 6). In a later library function (free_internal), one of the values stored by the malloc function is retrieved. However, because the value was stored in erroneous address (due to the fault), an incorrect value is loaded, which causes the program to crash at PC 0x404c70 (corresponding to node 19674). Consequently, the real CD for this fault is 3900 (19674 to 15775), although the IPS cardinality is only 11 nodes (10 nodes in the malloc function and the crash node). Thus, the fault has a high CD

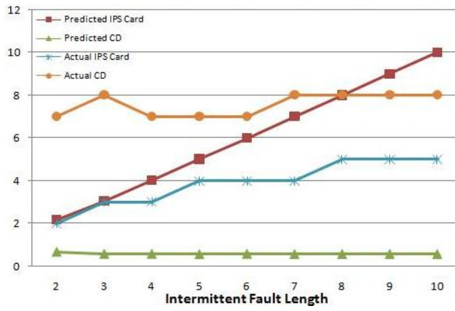

Fig. 4. The relationship between the intermittent fault length and average crash distance (CD) and the intermittent propagation set (IPS) for Print_Tokens.

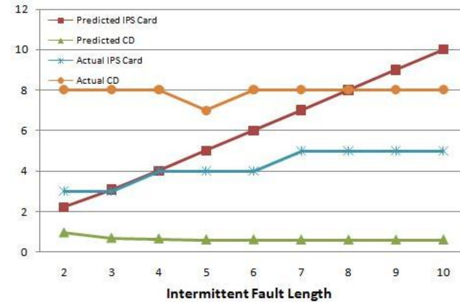

Fig. 5. The relationship between the intermittent fault length and average crash distance (CD) and the intermittent propagation set (IPS) for Replace.

and low IPS cardinality.

TABLE X
EXAMPLE CODE FRAGEMENT WITH HIGH CD AND LOW IPS CARDINALITY.

```
0x403fc8 <malloc+330> bne $v0[2],$zero[0],0x403f60
0x403fe0 <malloc+348> bgez $v0[2],0x403ff0
0x403f70 <malloc+2d8> lw $v1[3],15600($v1[3])
0x403f78 <malloc+2e0> sllv $v0[2],$a0[4],$s1[17]
0x403f80 <malloc+2e8> addu $v0[2],$a2[6],$v0[2]
0x403f88 <malloc+2f0> sw $v1[3],0($v0[2])
0x403f90 <malloc+2f8> sw $a1[5],4($v0[2])
0x403f98 <malloc+300> sw $v0[2],0($a1[5])
0x403fa0 <malloc+308> lw $v1[3],0($v0[2])
0x403fa8 <malloc+310> beq $v1[3],$zero[0],0x403fb8
. . .
0x404c50 <free_internal+630> addu $v1[3],$zero[0],$a3[7]
0x404c58 <free_internal+638> addiu $a2[6],$zero[0],1
```

E. Understanding the Effect of Intermittent Fault Length

In this subsection, we study the relationship between the intermittent fault's length and its predicted and actual CD and IPS cardinalities. Our goal is to understand how the DDG model's predictions are affected by the length of the fault. The numbers reported in this section pertain to the intermittent faults that are expected to cause crashes by the DDG model and actually do so in SimpleScalar (i.e., 74% to 81% of crashes in Table VI). We focus on faults whose CDs are less than 200 nodes since such faults constitute 90% of the total faults. The remaining faults are outliers and are hence ignored in this section. For the faults whose CD is less than 200, we plot the average CDs and IPS cardinalities (obtained in DDG and SimpleScalar), as a function of the fault's length. We report the results for only two applications, replace and print_tokens. Similar results were obtained for the other applications and, for brevity, are not reported. Note that each point in the graph represents the average CD or IPS values for all crash-causing faults of a given length. The main observations for the graphs in Figures 4 and 5 are as follows:

- The predicted and actual CDs remain relatively constant with the length of the intermittent fault. However, actual CD is consistently larger than expected CD, which reaffirms the earlier observation that DDG under-predicts the CDs (Section IV-D). Nonetheless, the observed deviation in CDs is less than 8 nodes on average for all faults independent of their lengths.
- Both predicted and actual IPS cardinalities steadily increase with respect to the fault's length. However, our analysis using the DDG model over-predicts the cardinalities compared to the actual values. Further, the predicted IPS values increase linearly with the intermittent fault length. This is because the IPS predicted by the DDG contains at least the nodes that are directly affected by the intermittent fault.
- The average actual CD is 7 to 8 nodes and is independent of the fault's length. Hence, most crashes occur within 8 nodes from the start of the intermittent fault. Therefore, we focus on intermittent faults that are at most 10 instructions long.

These results demonstrate that the DDG model's predictions of relationships between the intermittent faults' properties (CD and IPS) and the faults' lengths have similar trends as the corresponding values obtained with fault-injection experiments using SimpleScalar.

V. RELATED WORK

Fault avoidance and tolerance techniques: Fault-avoidance techniques attempt to minimize process variations [4], regulate voltage [7] or manage temperature [8]. Although fault-avoidance techniques reduce the base rate of intermittent faults, many faults still occur and escape to the software [6]. Fault tolerance techniques for intermittent faults require circuit-level changes [18], or often incur very high overheads even for fault-free devices [5].

Wells et al. [6] propose to recover from intermittent faults in software by suspending the faulty core and using a virtualization layer to manage over-committed systems. However, they assume that hardware circuits are used to detect intermittent faults. These circuits incur power and area overhead. To the best of our knowledge, there is no technique to mitigate intermittent faults using software alone. We evaluate the propagation of intermittent faults in programs as a first step in building software-only mechanisms to mitigate such faults.

Fault propagation studies: Gracia et al. [19] study the behavior of intermittent faults in a 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 error propagation studies have been performed for permanent faults [10] and transient errors [9], [20], to our knowledge no such study has been performed for intermittent faults.

The DDG model: The DDG model has been used to derive information about programs in prior work. These studies include: (1) Agrawal and Horgan [11] for slicing programs, (2) Austin and Sohi [12] for parallelizing programs, (3) Smolens et al. [20] for bounding error detection latency due to transient faults and (4) Pattabiraman et al. [17] for placing error detectors to detect transient faults. To the best of our knowledge, our work is the first to utilize the DDG to infer the effects of intermittent faults on programs.

VI. Conclusion

We studied the impact of intermittent hardware faults at the program level by gathering a dynamic instruction trace of fault-free program execution and analyzing this information using Dynamic Dependency Graph (DDG). We found that the majority of intermittent faults results in crashes (95% of non-benign faults) and that the DDG model is accurate in predicting the propagation of errors (within 100 nodes) for over 90% of crash-causing errors. Further, the DDG model took only from 4 to 89 seconds to analyze the propagation of intermittent faults in programs, whereas an equivalent fault-injection campaign using the SimpleScalar simulator took from 6 to 70 hours.

Acknowledgments

This research was supported by the National Science and Engineering Research Council through the Discovery Grant Program (Pattabirman, Gopalakrishnan) and by the Alexander Graham Bell Canada Graduate Scholarship (Rashid).

References

[1] C. Constantinescu, "Intermittent faults and effects on reliability of integrated circuits," *Proceedings of the Annual Reliability and Maintainability Symposium*, pp. 370–374, 2008.

[2] J. McPherson, "Reliability challenges for 45nm and beyond," *ACM IEEE Design Automation Conference*, pp. 176–181, 2006.

[3] 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, pp. 183–190, 2002.

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

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

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

[7] 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," *International Symposium on Microarchitecture*, pp. 7–18, 2003.

[8] 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, pp. 94–125, 2004.

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

[10] 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," *International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 265–276, 2008.

[11] H. Agrawal and J. Horgan, "Dynamic program slicing," *ACM SIGPLAN Notices*, vol. 25, no. 6, pp. 246–256, 1990.

[12] T. Austin and G. Sohi, "Dynamic dependency analysis of ordinary programs," *Annual International Symposium on Computer Architecture*, vol. 20, no. 2, pp. 342–351, 1992.

[13] D. Burger and T. Austin, "The SimpleScalar tool set, version 2.0," *Computer Architecture News*, vol. 25, no. 3, pp. 13–25, 1997.

[14] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria," *The International Conference on Software Engineering*, pp. 191–200, 1994.

[15] G. Saggese, A. Vetteth, Z. Kalbarczyk, and R. Iyer, "Microprocessor sensitivity to failures: Control vs execution and combinational vs sequential logic," *Dependable Systems and Networks, International Conference on*, vol. 0, pp. 760–769, 2005.

[16] W. Kao, R. Iyer, and D. Tang, "Fine: A fault injection and monitoring environment for tracing the unix system behavior under faults," *IEEE Transaction on Software Engineering*, vol. 19, no. 11, pp. 38–43, 1993.

[17] K. Pattabiraman, Z. Kalbarczyk, and R. Iyer, "Application-based metrics for strategic placement of detectors," *Pacific Rim International Symposium on Dependable Computing*, pp. 75–82, 2005.

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

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

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