One Bit is (Not) Enough: An Empirical Study of the Impact of Single and Multiple Bit-Flip Errors

Behrooz Sangchoolie[*], Karthik Pattabiraman[+], Johan Karlsson[*]
[*]*Department of Computer Science and Engineering, Chalmers University of Technology*
[+]*Department of Electrical and Computer Engineering, University of British Columbia*
Emails: {behrooz.sangchoolie, johan}@chalmers.se, karthikp@ece.ubc.ca

***Abstract*—Recent studies have shown that technology and voltage scaling are expected to increase the likelihood that particle-induced soft errors manifest as multiple-bit errors. This raises concerns about the validity of using single bit-flips for assessing the impact of soft errors in fault injection experiments. The goal of this paper is to investigate whether multiple-bit errors could cause a higher percentage of silent data corruptions (SDCs) compared to single-bit errors. Based on 2700 fault injection campaigns with 15 benchmark programs, featuring a total of 27 million experiments, our results show that single-bit errors in most cases yields a higher percentage of SDCs compared to multiple-bit errors. However, in 8% of the campaigns we observed a higher percentage of SDCs for multiple-bit errors. For most of these campaigns, the highest percentage of SDCs was obtained by flipping at most 3 bits. Moreover, we propose three ways of pruning the error space based on the results.**

***Keywords*—fault injection; transient hardware faults; single/multiple bit-flip errors; error space pruning;**

I. Introduction

Technology and voltage scaling is making transistors increasingly susceptible to soft errors caused by ionizing particles [1], which increase the rate of transient faults. These errors can degrade system reliability, by producing *silent data corruptions (SDCs)* causing unacceptable or catastrophic system failures. A cost-effective way of reducing the risk that hardware faults cause such failures is to introduce software-implemented error handling mechanisms [2], [3]. The effectiveness of these mechanisms is often evaluated by means of fault injection. Fault injection can be carried out either at the hardware or at the software level. The former is more accurate but is often slow and cumbersome. Software Implemented Fault Injection (SWiFI) has been widely used to emulate hardware errors in software. SWiFI is often faster than hardware injection, and requires no hardware support.

An important challenge in SWiFI techniques is the selection of the fault model, which needs to be both straightforward to implement, and representative of real hardware faults. The single bit-flip model has been a popular engineering approximation to mimic particle induced soft errors both in the combinational logic and storage elements (e.g., flip-flops). However, earlier studies have found that many soft errors that occur in the processor manifest as multiple-bit errors at the application level [4], [5], [6]. This observation has led researchers to question the validity of the single bit-flip model for representing transient faults due to soft errors. Therefore, the fault model should consider both single-bit and multiple-bit errors when calculating measures such as *error coverage* [7], [8] and *error resilience* [9], [10] (we focus on the latter).

The main question we ask is *"Does the multiple bit-flip model significantly differ from the single bit-flip model in terms of its impact on programs' error resilience, and if so by how much?"*. Prior work [10], [11], [12], [13] has studied the impact of double bit-flip errors on a program, i.e., injecting two errors in a single word or multiple words. These papers assume (without providing evidence) that single/double bit-flip errors are sufficient when measuring programs' error resilience. However, in this paper, we inject multiple bit-flips systematically to provide *evidence* regarding the number of bit-flips needed to cause pessimistic SDC results. There exists very little work on studying the effects of multiple bit-flip errors beyond double bit-flips due to two challenges.

First, there is no commonly agreed model to map transient faults, caused due to soft errors, to their software-level manifestation. In fact, it may not even be possible to find such a representative model. This is why in this paper, we propose a systematic error space exploration that is based on *error space clustering*, where each cluster is represented by two parameters, (i) the number of bit-flip errors that could occur in the cluster; and (ii) the distance (in terms of the number of dynamic instructions) between consecutive injections. Using these parameters, we form 180 clusters for each program and conduct multiple bit-flip fault injection experiments. *To the best of our knowledge, we are the first to study the effect of multiple bit-flip errors on programs beyond double bit-flips, through the use of clustering techniques.*

Secondly, the space of multiple bit-flip errors is extremely large, and conventional techniques for reducing (i.e., *pruning*) the error space may not be applicable. Further, almost all the existing techniques for pruning the error space [14], [15], [16] work with the single-bit fault model, and are not easily extensible to multiple-bit errors. Therefore, in this paper, we also propose three ways of pruning the error space based on the fault injection results obtained. First, we find that it is unnecessary to expose a program to a very high number of bit-flip errors, as in this case, only a small fraction of the injected errors are *activated*. Second, we identify programs where the single bit-flip model causes pessimistic (i.e., conservative) percentage of SDCs compared to when the multiple bit-flip model is used. For these programs, the results of multiple bit-flip injections can be replaced by single bit-flip fault injection results. Third, we use the single bit-flip fault injection results to prune the error space of multiple bit-flip campaigns by targeting only a fraction of these errors, based on where the first error should be injected. Together, the three techniques allow us to prune the space of multiple-bit fault injections.

In summary, the paper makes the following contributions:

- Extends an LLVM (Low Level Virtual Machine)-based fault injector that injects single bit-flips at the LLVM compiler's intermediate code level [17] (§IV-A), to inject multiple-bit errors (§III-C) in a single word (§IV-B) as well as multiple words (§IV-C).
- Performs more than *27 million* experiments (§III-E) on 15 benchmark programs (§III-D) and for 182 single/multiple bit error configurations (§III-C) using two different fault injection techniques (§III-A).
- Quantifies the maximum (upper bound) number of multiple bit-flip errors needed to cause pessimistic percentage of SDCs (§IV-B and §IV-C2). We find that the single bit-flip model mostly (92% of all campaigns) results in pessimistic percentage of SDCs compared to the multiple bit-flip model; and even when it does not, in most cases, at most three errors are enough to result in a pessimistic percentage of SDCs.
- Derives new insights about how the results of single bit-flip experiments can be used to prune the multiple bit-flip error space by targeting only a fraction of these errors, that reveal weaknesses of the programs under test (in terms of the number of SDCs) that are not revealed by the single bit-flip model (§IV-C3). We find that single bit-flip experiments that result in an SDC or program crash, contributing to around 27-100% of the experiments, can be pruned by the derived insights.

II. Fault Model and Background

A. Fault Model

In this paper, we use the *bit-flip* model to mimic transient faults due to soft errors that occur in the processor's register file, ALUs, and in different pipeline registers that eventually manifest as a data corruption in a source/destination register. The bit-flip model has been also used in other related work [2], [18], [19] as a model for transient faults caused by soft errors. Unlike these works, our model includes both *single* and *multiple bit-flip* errors. Similar to prior work [4], [5], we do not consider faults in memory since ECC protects the memory against single/double bit-flips. ECC is however, incapable of protecting the memory against multiple bit-flips in the same word. However, in this paper we mainly focus on multiple bit-flips in multiple words, which may be detected by ECC as they may manifest as single/double bit-flips in the same word.

B. Error Coverage vs. Error Resilience

Fault injection techniques have been extensively used to evaluate the effectiveness of error handling mechanisms as well as to improve the accuracy of measures such as error coverage (c) [7], [8]. Error coverage is defined as the conditional probability that the program recovers, given the occurrence of a fault, and consists of both recovery from crashes and SDCs. An SDC occurs when the program terminates normally, but the output is erroneous. In practice however, SDCs are the more important class of failures as the erroneous outputs are generated with no indication of failure, making them very difficult to detect. Therefore, instead of the error coverage, we use *error resilience* [9], [10] as the dependability metric. Error resilience is defined as the conditional probability that the program does not produce an SDC after a transient hardware fault occurs and impacts the program state (i.e., similar to work such as [20], [21], [22] it deals with faults passing the hardware and seen by the software). The error resilience and similar metrics such as error sensitivity [11], [23] are used to evaluate the effectiveness of error handling mechanisms.

C. Related Work

Traditionally, most fault injection studies at the program level have focused on the single bit-flip model, i.e., injecting single bit-flips into programs. However, recently, there have been some studies focusing on double bit-flip model [10], [11], [12]. Lu et al. [10] compare the results of injecting single bit-flip errors with injecting double bit-flip errors in a single word and in different words at the LLVM compiler's intermediate code level using the LLFI [24] fault injector. They find that there is not much variation between the error resilience of the different models. The main focus of the work is on the fault injection tool rather than a thorough study of the impact of multiple bit-flip errors. Ayatolahi et al. [11] compare the single bit-flip model with the double bit-flip model at the assembly-level code. In their study, double bit-flip errors are only injected into a single word (i.e., register or memory location). They also find that the SDC results obtained for the two fault models are only marginally different. Adamu-Fika and Jhumka [12] compare the results of injecting double bit-flip errors in a single word with those of injecting into different words at the LLVM compiler's intermediate code level. Similar to the other two studies, the results of their experiments show that, on average, the difference between the percentage of *data failures* for the two models is marginal. However, they do not consider the relative positions of the faults injected, nor do they generalize their findings beyond double bit-flips.

Compared to the above mentioned studies, in this paper, we go beyond the double bit-flip model by injecting up to 30 bit-flip errors in single words as well as different words in each program run. We also consider a wide range of parameters that may influence the fault injection results and characterize the space thoroughly. This way, we can analyze the sensitivity of the results with respect to the fault injection parameters used. Since the multiple bit-flip error space is significantly large, in this paper, we also derive insights on how the error space could be further pruned from our results, which also distinguishes our study from the earlier studies.

There has also been little work targeting programs with multiple errors [25], [26]. Jiantao Pan [25] introduces a model called *dimensionality* to pin-point the number of function call parameters that are responsible for a failure. The model is used in a subsequent work [26] to improve software robustness. However, compared to our fault model, the dimensionality model has two main limitations; *(i)* multiple errors are only introduced to the parameters of each interface, which may not be representative of multiple errors that occur in variables used within the function; *(ii)* the number of errors that are introduced in each interface is limited by the number of parameters used by the interface.

There are also studies addressing intermittent faults, which could model multiple-bit errors. Intermittent faults are those that show up intermittently at the program level. For example, Rashid et al. [27] build an intermittent fault model at the microarchitectural level using stuck-at-last-value and stuck-at-zero/one models. However, they assume that *(i)* a microarchitectural unit may be affected by at most a single intermittent fault and *(ii)* at most one microarchitectural unit may be affected by an intermittent fault. These assumptions may not hold for transient faults due to soft errors, which is our focus.

D. Error Clustering and Error Sampling

The error space under the single bit-flip model is dependant on the number of register bits available in a target system. Unfortunately, the high number of bits in a typical program makes the error space prohibitively large. For example, assume that each instruction reads or writes only one register; let d be the number of dynamic instructions in a program and b be the number of bits in a register, then $d * b$ would be the size of the single-bit error space. This makes it infeasible to conduct exhaustive fault injection campaigns for workloads with a high number of dynamic instructions. This is why prior work has either randomly sampled the error space, or used error clustering to find classes of equivalent errors which could then be pruned to facilitate exhaustive fault injection campaigns [14], [15], [16]. However, all of these papers have focused on single-bit errors, and hence their heuristics for sampling and clustering are specific to the single bit-flip scenario.

Conducting multiple-bit injections adds another dimension to the (already large) error space, making it even more necessary to use error space pruning techniques. For example, let m be the number of multiple-bit errors in one run of a program where the maximum number of errors is bounded by $d * b$, then the error space could be as big as $\sum_{m=2}^{d*b}(d * b)^m$. This makes it infeasible to conduct exhaustive fault injection campaigns even for workloads with a fairly low number of dynamic instructions; this is why in this paper, we use clustering in the context of multiple-bit errors to explore the error space in a more systematic way (by placing errors with similar characteristics in the same error class). Moreover, the error clusters as well as various heuristics that are specific to multiple-bit errors are used to prune the error space by finding a class of errors that leads to pessimistic percentage of SDCs.

III. Experimental Setup

In this section we first present the different fault injection techniques used in §III-A. Then in §III-B, we present the fault injection tool used in the paper and in §III-C we present our extensions to it. In §III-D, we present the benchmark programs used in our experiments. In §III-E, we present the design of experiments, and how we classify the outcome of each experiment. Finally, in §III-F, we present the research questions related to error space understanding and pruning.

A. Fault Injection Techniques

In this paper, we conduct our fault injection experiments using two techniques, namely *inject-on-read* and *inject-on-write*. Using these techniques, faults are only injected in live registers, which eliminate faults with no possibility of activation. The motivation for injecting faults in live registers is that 80-90% of randomly injected faults are often not even activated [28], [29]. Examples of these are faults placed in a register just before the register is written into (and is overwritten), and faults that are injected into unused registers.

1) Inject-on-read: This technique only injects a fault into a register just before it is read by an instruction [16], [23], [30]. Using this technique, Barbosa et al. [16] managed to reduce the error space of workloads by two to five orders of magnitude. The inject-on-read is well suited for emulating errors that propagate into a register, for example due to a direct hit by an ionizing particle. In this technique, all faults targeting a specific bit of a given register, from the time the register is written into until it is read, are considered equivalent.

Note that to obtain an accurate estimation of different dependability measures, it would be necessary to apply a weight factor corresponding to the number of faults in each equivalence class [16], [23], [31]. However, the aim of this paper is to compare the single and multiple bit-flip models, rather than to find an absolute dependability measure for programs. Therefore, we do not apply such a weight factor.

2) Inject-on-write: This is a technique that is used to reduce the error space size by only injecting an error into a register right after it is written into by an instruction [23], [24], [32]. It aims to mimic faults in computation, such as the ones that occur in the arithmetic logic units (ALUs) and in different pipeline registers that eventually manifest as an error in a destination register.

B. LLFI Fault Injection Tool

In this paper, we use LLFI [24], an open source fault injector, that injects faults into the LLVM [17] framework's intermediate code of a program. LLVM is a collection of reusable compiler tools and components, and allows analysis and optimization of code written in multiple programming languages. The key component of LLVM is its intermediate representation (IR), an assembly-like language that abstracts out the hardware and ISA-specific information. LLFI has been used in several other work [9], [10], [12], for injecting single and double bit-flip errors using inject-on-read and inject-on-write techniques. In this work, we have extended LLFI to facilitate the injection of multiple bit-flip errors as explained in the next section[1].

[1]LLFI is available at http://github.com/DependableSystemsLab/LLFI

C. Extending LLFI for Multiple Bit-Flip Injections

LLFI [24] defines single bit-flip errors as time-location pairs according to a fault-free execution of a program. The location is selected from IR registers, and the time corresponds to a dynamic IR instruction. To model multiple bit-flip errors, we extend the time-location parameters by two additional parameters, namely *max-MBF* and *win-size*, which allow us to cluster the error space into different classes of errors to be able to explore the error space in a more systematic way. The max-MBF parameter controls the number of bit-flip errors that could occur in one run of a program. Selecting a certain value, say 5, as the max-MBF does not necessarily mean that five errors will be injected into the program. This is because the program may crash prematurely (after the first injection, say), causing the remaining faults to not be injected. Therefore, max-MBF is in fact, the *maximum* number of bit-flip errors that occur in the program. The win-size, on the other hand, controls the number of dynamic instructions that should be executed between consecutive injections. For example, if the win-size is equal to 2, the dynamic instruction distance between each injection is equal to two.

As there are no commonly agreed values for the new parameters in the literature, we consider a wide variety of values for the parameters when studying the impact of multiple bit-flip errors on programs. These value ranges cover various multiple bit-flip scenarios temporally, enabling us to perform sensitivity analysis. In this paper, we use 10 different values for the max-MBF (see Table I) ranging from 2 to 30. We motivate the use of 30 as a max-MBF value in §IV-C1.

For the window size parameter, we select nine win-size values covering dynamic window sizes from zero to 1000 (see Table I). A window size of zero implies that the injections, following the first one, will be performed into the same instruction (i.e., register). The rationale behind limiting the maximum value of this parameter to 1000 is that we predominantly consider multiple bit-flip errors in software that are caused by a single transient fault in the processor. Such faults are likely to affect instructions that are "in-flight" in the processor's instruction window[2]. Typical instruction windows in modern processors are a few hundred of instructions in size, and hence 1000 is a reasonable upper bound. Six of the values selected are constants (0, 1, 4, 10, 100, 1000). The remaining three values are randomly selected from a range of 2-10, 11-100, or 101-1000, to achieve better representativeness.

The chosen win-size values could also represent multiple unconnected transient faults that cause errors in instructions that are apart from each other by less than 1000 dynamic instructions. However, it is very unlikely that multiple transient faults (due to multiple soft errors) occur in a single run of a program, that too within a short time of each other.

D. Benchmark Programs

We target 15 programs in our set of fault injection experiments. We select a diverse set of programs with respect to source code implementation, code size, input type/size, functionality, etc. from two distinct benchmark suites, namely MiBench [33] and Parboil [34] (see Table II).

1) MiBench Benchmark Suite: This benchmark suite contains a set of commercially representative embedded programs. The programs are placed into six different packages of automotive, consumer, network, office, security, and telecomm. In this paper, we select 11 programs from these packages (see Table II). MiBench provides two inputs for every program, namely *small* and *large*. We use the small inputs in our set of experiments as we need to perform thousands of fault injection experiments and hence need inputs that do not lead to long running times.

2) Parboil Benchmark Suite: This benchmark suite contains a set of programs selected from scientific and commercial fields. We select four programs from this benchmark suite (see Table II). Two of them (bfs and histo) are taken from the *base* implementation package. The remaining two programs (sad and spmv) are from the *CPU* implementation package.

Table II also shows the total number of candidate instructions for inject-on-read and inject-on-write fault injection techniques. From the table, we can see that the number of instructions that are available for inject-on-read is higher than the inject-on-write. This is because instructions, such as the `store` instruction, do not have destination register in the LLVM IR; thus they are not selected as candidates for fault injection in inject-on-write.

E. Experimental Design and Outcome Classification

We conduct 182 *fault injection campaigns* for each of the benchmark programs presented in §III-D. A fault injection campaign refers to a set of fault injection experiments using the same fault model on a given *workload*; a workload is a program running with a given input. Half of the campaigns use the inject-on-read technique presented in §III-A1, whereas the other half use the inject-on-write technique presented in §III-A2. In addition to two single bit-flip campaigns, each using a fault injection technique, we perform multiple bit-flip errors using the parameters (max-MBF, win-size) in Table I.

Table I
VALUES SELECTED FOR THE MAXIMUM NUMBER OF MULTIPLE BIT-FLIP ERRORS (MAX-MBF) AND THE DYNAMIC WINDOW SIZE (WIN-SIZE) BETWEEN CONSECUTIVE INJECTIONS.

max-MBF index	max-MBF value	win-size index	win-size value
m1	2	**w1**	0
m2	3	**w2**	1
m3	4	**w3**	4
m4	5	**w4**	random between 2-10
m5	6	**w5**	10
m6	7	**w6**	random between 11-100
m7	8	**w7**	100
m8	9	**w8**	random between 101-1000
m9	10	**w9**	1000
m10	30		

[2]The instruction window is the set of all instructions that have been decoded but not yet committed in a superscalar processor.

Table II
SELECTED BENCHMARK PROGRAMS

Benchmark	Package	Program (LoC)	Total number of candidate instructions for fault injection		Description & Input
			inject-on-read	inject-on-write	
MiBench	automotive	basicmath (178)	3,683,881	2,964,600	Performs mathematical calculations such as cubic equation calculation and square root calculation on a set of constants.
		qsort (35)	2,615,557	2,214,245	Implements the Quick Sort algorithm on a list of words.
		susan_corners(1700)	2,449,209	2,088,322	Finds corners of a black & white image of a rectangle.
		susan_edges(1700)	5,188,476	4,413,577	Finds edges of a black & white image of a rectangle.
		susan_smoothing(1700)	62,752,639	49,105,460	Smooths a black & white image of a rectangle.
	telecomm	FFT (215)	5,313,377	4,526,716	Performs Fast Fourier Transformation on an array of data.
		IFFT (215)	5,423,988	4,620,938	Performs reverse FFT on an array of data.
		CRC32 (107)	28,746,216	23,270,737	Implements the 32-bit Cyclic Redundancy Check on a sound file.
	network	dijkstra (133)	67,617,629	54,495,536	Uses Dijkstra's algorithm to find the shortest path between pairs of nodes constructed from an adjacency matrix representation graph.
	security	sha (188)	30,609,559	25,726,389	Implements the well known SHA (secure hash algorithm), generating a 160-bit digest from an ASCII text file.
	office	stringsearch (340)	161,533	114,835	Searches for words in phrases using case insensitive comparison.
Parboil	base	bfs (592)	113,582,521	94,021,100	Uses the breadth-first search algorithm to compute the shortest-path cost from a single node to every reachable node in an irregular graph of uniform edge weights derived from the map of New York.
		histo (610)	678,224,521	566,829,877	Computes a 2-D saturating histogram with a maximum bin count of 255 of the default input set.
	cpu	sad (944)	648,604,565	510,295,230	Calculates the sum of absolute differences in the default input set.
		spmv (619)	11,003,882	8,965,172	Computes the product of a sparse matrix with a dense vector. We select the small input, which is a sparse matrix in coordinate format.

Each campaign consists of 10,000 fault injection experiments to obtain tight error bounds. *Thus, we perform a total of* $10,000 * 182 * 15 = 27,300,000$ *experiments.* We also compute error bars at the 95% confidence intervals.

The outcome of each experiment is classified into one of the following categories:

- *Benign.* The program terminates normally and the injected error does not affect the program's output. This category could be the result of internal robustness of the program and it contributes to overall error resilience.
- *Detected by Hardware Exceptions.* The injected error raises a hardware exception. Almost all these exceptions cause the program to *crash*, however there are very few cases where a hardware exception is raised without causing a crash. Errors detected by hardware exception mechanisms contribute to the overall error resilience, as the program could potentially call a recovery routine and prevent the program from producing an erroneous result. These exceptions include *segmentation faults* (accessing memory words outside the legal memory segment boundary), *aborts* (programs aborted by themselves or the OS), *misaligned memory accesses* (memory accesses are not aligned at four bytes), and *arithmetic errors* such as division by zero.
- *Hang.* The program fails to terminate within a predefined time, which is set by LLFI to be one or two orders of magnitude greater than the execution time of the fault-free run of the program. Errors that result in this category also contribute to the overall error resilience as watchdog timers could be used to detect them.
- *NoOutput.* The program terminates, without generating an output. Errors that result in this category also contribute to the overall error resilience as there is an indication that the program needs to be executed again.
- *Silent Data Corruption (SDC).* The program terminates normally, but the output is incorrect (based on a bit-wise comparison), and there is no indication of the failure.

As mentioned above, the first four outcome categories (Benign, Detected by hardware exceptions, Hang, and NoOutput) contribute to the error resilience. Recall that the error resilience is defined as the probability that the program does not cause an SDC, which is why we focus on the SDC outcome category. Among the four error resilience categories, the Benign category is the result of internal robustness of the program, while the other three categories correspond to when an error is detected - we refer to them as *Detection.*

F. Research Questions

In this section, we present the research questions that are investigated in this paper. The research questions are motivated by the three error pruning techniques we investigate.

The first error space pruning layer deals with the selection of an upper bound for the max-MBF parameter, since there is no commonly agreed mapping model that could be used to reason about the number of software-level errors due to a hardware transient fault. Though we choose 30 as an upper bound for the max-MBF, the actual number of activated errors may be far fewer allowing us to prune the multiple error injection space. So the first research question deals with the number of errors that are actually activated when multiple errors are injected and do not result in program crashes.

- ***RQ1.*** When multiple errors are injected, how many errors are activated before the program crashes (if it crashes)?

In the second layer of error space pruning, we classify fault injection results with respect to parameters such as the fault injection technique used (inject-on-read and inject-on-write), the maximum number of bit-flips injected (max-MBF), and the dynamic window size between consecutive injections (win-size), to investigate whether we could further prune the error space by finding parameter values that result in pessimistic percentage of SDCs (i.e., conservative upper-bounds). Therefore, we ask the following research questions:

- ***RQ2.*** Does the single bit-flip error model result in pessimistic percentage of SDCs when compared with the multiple bit-flip error model?
- ***RQ3.*** Is there an upper bound to the maximum number of multiple bit-flips needed to cause pessimistic percentage of SDCs?
- ***RQ4.*** Is there a maximum dynamic window size that causes pessimistic percentage of SDCs?

Using the results obtained from the second layer of error space pruning, the multiple bit-flip error space could be significantly pruned allowing us to only focus on a certain subset of the max-MBF and win-size parameters. However, depending on the size of the program, conducting fault injection experiments even on the pruned error space may still be very time-consuming. This is why in the third layer of error space pruning, we ask the following question:

- ***RQ5.*** Is it possible to find fault injection locations that are insensitive to multiple bit-flip errors compared to single bit-flip errors, and exclude them from the multiple-injection error space?

IV. Experimental Results

In this section, we present detailed classifications of fault injection results with respect to the parameters, max-MBF and win-size as well as the type of fault injection technique used. These classifications help us quantify the differences between candidate values that can be chosen for each parameter, which allows us to answer the research questions presented in §III-F. We start by understanding the outcomes of the single-bit fault injection experiments (§IV-A), followed by multiple injections into the same register (§IV-B), and then finally multiple fault injections in different registers (§IV-C).

A. Results for the Single Bit-Flip Model

In this section, we present the results of fault injections using the single bit-flip model to serve as a baseline for comparison with the multiple bit-flip injections. Fig. 1 shows the outcome classification results with the single bit-flip model. Fig. 1a and Fig. 1b show the results for when inject-on-read and inject-on-write fault injection techniques are used, respectively. Recall that the Detection category is the sum of the results for *Hang*, *NoOutput* and *Detected by Hardware Exception* categories. The percentage of experiments classified as *Hang* and *NoOutput* is insignificant (less than 0.3%), and hence most of the experiments in the Detection category were detected by hardware exceptions.

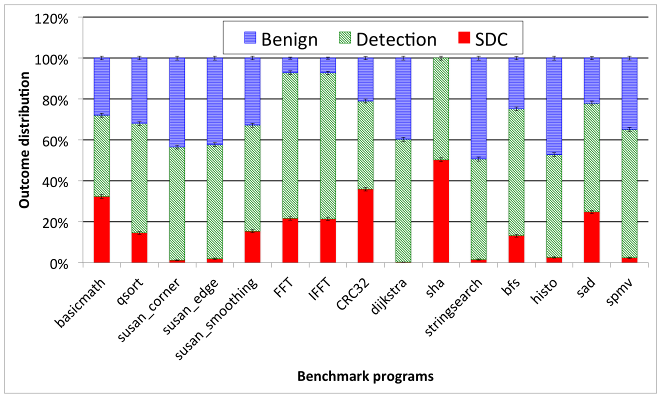
Fig. 1 shows that overall, the SDC percentage when using the inject-on-write technique is higher than that when using the inject-on-read technique. A similar trend was also observed by Sangchoolie et al. [23]. The reasons for this difference are *(i)* the type of data-items stored in source/destination registers as well as *(ii)* the number of times that these registers are accessed throughout the execution of the program. Registers could hold data-items of different types such as memory addresses, data variables, and control information. Errors injected in memory addresses are mostly detected by hardware exception mechanisms, causing a higher percentage of crashes and hence lower percentage of SDCs [9], [23]. Both source registers and destination registers could hold a memory address, however, an address may be read multiple times after it is written into. This increases the probability of an error being injected into an address when using the inject-on-read technique, which would eventually result in a lower percentage of SDCs for the results obtained using the inject-on-read technique compared to the inject-on-write technique.

B. Results When Targeting Multiple Bits of the Same Register

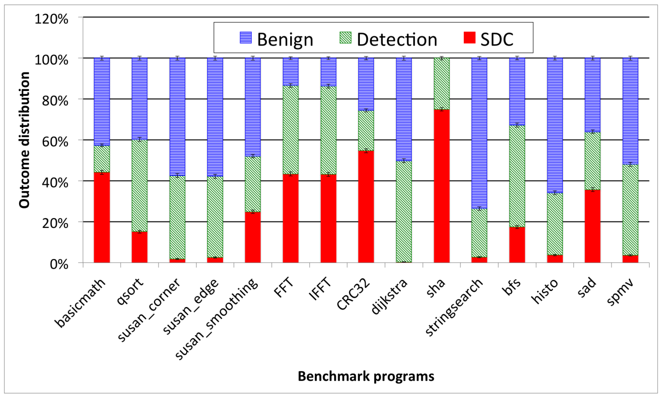
Fig. 2 shows the classification of fault injection results when the multiple injections are performed into the same instruction (i.e., register). In other words, for each program, the dynamic window size (win-size) value is zero, and only the max-MBF parameter is varied from 1 (the leftmost bar) to 30 (the rightmost bar). The goal of this experiment is to understand how much the max-MBF parameter alone contributes to the percentage of SDCs.

Fig. 2a and Fig. 2b show the results for when inject-on-read and inject-on-write fault injection techniques are used, respectively. The leftmost result bar for each benchmark program represents the percentage of SDCs when only a single-bit error is injected, while the other result bars correspond to the percentages of SDCs caused by different numbers of multiple bit-flip errors ranging from 2 to 30.

Fig. 2 shows that for the majority of the programs, the SDC results obtained for the single bit-flip model is either pessimistic, or very close to the ones obtained for the multiple bit-flip model. However, for basicmath and CRC32 programs, the SDC results due to the single bit-flip model are significantly lower (especially when using the inject-on-write) than the results obtained for the multiple bit-flip model, and hence the single bit-flip model does not yield pessimistic SDC results for these programs. This behaviour can be explained by looking at Fig. 1, where we see that single bit-flip errors injected into these programs result in the lowest percentage of Detections

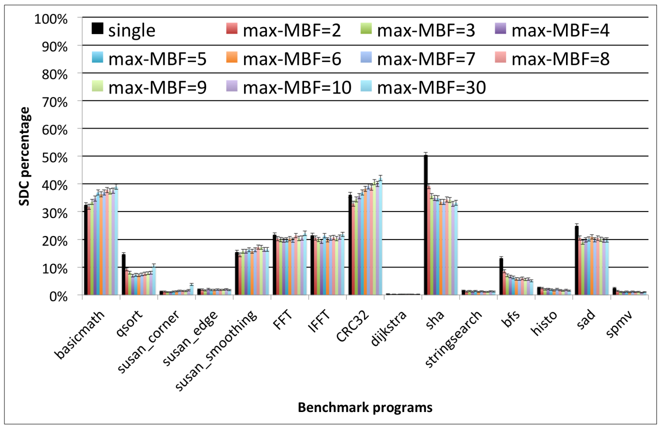


(a) inject-on-read

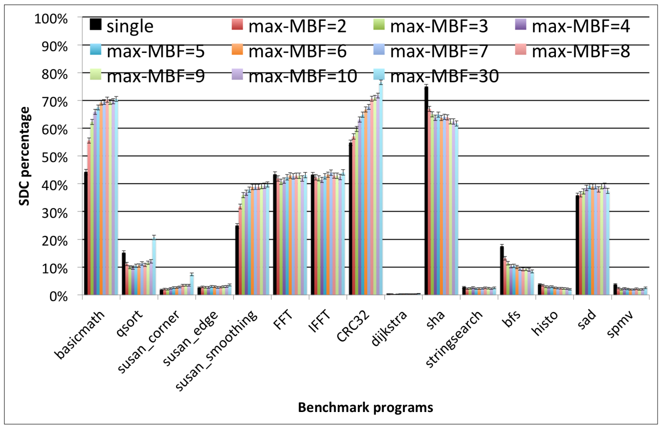


(b) inject-on-write

Figure 1. Fault injection outcome classification for campaigns using single bit-flip model. The Detection category refers to the sum of Detected by Hardware Exception, Hang and NoOutput categories. The error bars indicate 95% confidence intervals.



(a) inject-on-read



(b) inject-on-write

Figure 2. Percentage of SDCs for injecting different number of errors into the same instruction/register (i.e., win-size = 0). The leftmost and rightmost bars, for each program, represents the percentage of SDCs when injecting 1 and 30 errors, respectively. The error bars indicate 95% confidence intervals.

when compared with the other programs. This implies that there are fewer possibilities of hardware exceptions to be raised due to the injection of errors in these programs. Therefore, many of the errors injected remain undetected, thereby resulting in a higher percentage of SDCs.

For qsort and susan-corner programs, the single bit-flip model results in pessimistic percentage of SDCs compared to the multiple bit-flip model, except for the case when max-MBF $= 30$. However, it is unlikely that these number of bits are affected by a single fault; this configuration (max-MBF $= 30$) is mainly selected for answering RQ1. Therefore, for the qsort and susan-corner programs also, the single bit-flip fault model provides us with a pessimistic estimate of the percentage of SDCs caused due to multiple bit-flip error injections.

***RQ2-Answer*:** For the majority of the benchmark programs, the results obtained for the single bit-flip model is either pessimistic or very close to the ones obtained for the multiple bit-flip model for bit-flips in the same register (i.e., $win - size = 0$).

C. *Results When Targeting Bits of Multiple Registers*

In this section, we consider multiple bit-flips in multiple registers accessed by different instructions. To control the distance between consecutive injections, we choose the dynamic window sizes (win-size) that are greater than zero (win-size>0) from Table I. We first attempt to bound max-MBF by studying how many errors are activated when the max-MBF=30 (§IV-C1). We find that only a small fraction of these errors are activated, making it unnecessary to select higher values for max-MBF. However, as the error space is still large, we search for max-MBF/win-size pairs in the space that cause pessimistic percentage of SDCs (§IV-C2). Finally, we investigate whether the single bit-flip fault injection results can help prune the multiple bit-flip error space (§IV-C3).

1) Number of Activated Errors: Fig. 3 shows the distribution of the number of activated errors before causing a program to crash, given that we intend to inject 30 bit-flip errors. The reason for selecting such a high value (max-MBF=30) is to find the portion of errors that could remain undetected and can hence be pruned. Note that the results presented here include all win-size values shown in Table I.

Fig. 3 shows that at most five activated errors are enough to cause a program to crash in more than 96% (78%) of the experiments using inject-on-read (inject-on-write) techniques. Furthermore, 3% and 14% of the inject-on-read and inject-on-write experiments, respectively, managed to activate six to ten errors. And finally, only around 1% of the inject-on-read experiments and 8% of the inject-on-write experiments had more than 10 activated errors. Thus, we see that an upper bound of 10 errors for max-MBF is sufficient to capture the majority of fault injection outcomes, and hence can be used to bound the value of max-MBF. As one could expect, the errors that remain undetected would most likely result in SDCs.

***RQ1-Answer*:** Around 99% of inject-on-read and 92% of inject-on-write experiments had fewer than 10 activated errors.

2) Max-MBF/win-size Pairs that Cause Pessimistic Percentage of SDCs: In the previous section, we studied the effects of the max-MBF on the number of activated errors in the program. We now examine the effects of the max-MBF on the SDC percentages. Fig. 4 and Fig. 5 show the SDC results for the experiments targeting bits of multiple registers using the inject-on-read, and inject-on-write techniques, respectively. Both of these figures show that when increasing the number of bit-flip errors, the general trend for the SDC results is declining, regardless of the value of win-size selected. We further study each technique in detail below.

a) Results for the inject-on-read Technique: Fig. 4 shows the SDC results for the experiments targeting bits of multiple registers using the inject-on-read technique. The 95% confidence intervals for these results are between ± 0.19 for dijkstra and ± 0.97 for sha. According to this figure, in 13 programs, the percentage of SDCs caused due to the single bit-flip model is higher than or almost the same as (i.e., difference less than one percentage point) the ones caused due to the multiple bit-flip model. However, for 2 programs (CRC32 and stringsearch), there are multiple bit-flip campaigns that result in a higher percentage of SDCs. Even for the 2 programs, the percentage of SDCs caused due to the single bit-flip model is only around two percentage points lower than the multiple bit-flip configuration that causes the highest percentage of SDCs. *Thus, the single bit-flip model provides a pessimistic upper-bound on SDCs for most of the programs.*

It is interesting to note that even when the single bit-flip model does not result in pessimistic SDC results, *two errors are enough to result in the highest (pessimistic) percentage of SDCs*, regardless of the value of win-size selected (see Table III). The value of this observation is that in the case of the inject-on-read technique, there is no need to perform more than two injections to estimate the error resilience of a system.

Fig. 4 also shows that except for a couple of programs such as CRC32 and susan-smoothing, there is not much variation between the percentage of SDCs obtained for different win-

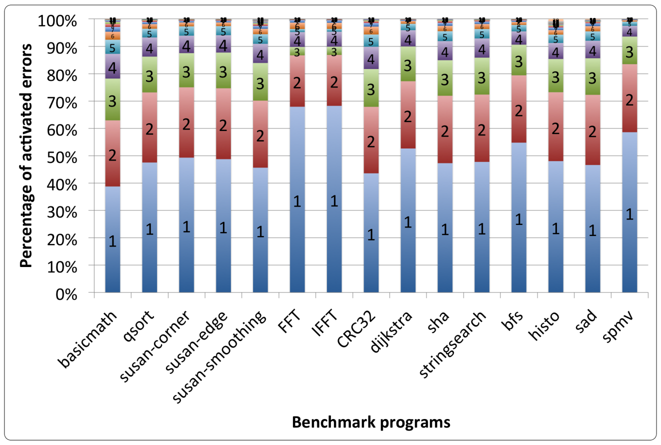
Table III
CONFIGURATIONS THAT RESULTED IN THE HIGHEST PERCENTAGES OF SDCS, AMONG ALL MULTIPLE BIT-FLIP ERROR CAMPAIGNS.

Program	**inject-on-read**		**inject-on-write**	
	max-MBF	**win-size**	**max-MBF**	**win-size**
basicmath	2	100	3	1
qsort	2	100	3	1
susan_corner	2	1000	4	1
susan_edge	2	1000	3	1
susan_smoothing	2	1000	3	1
FFT	2	1	2	1
IFFT	2	1	2	1
CRC32	2	100	2	100
dijkstra	2	4	3	4
sha	2	10	2	1
stringsearch	2	RND(2-10)	2	4
bfs	2	1000	2	1000
histo	2	RND(2-10)	6	1
sad	2	1000	2	4
spmv	2	1000	2	RND(11-100)

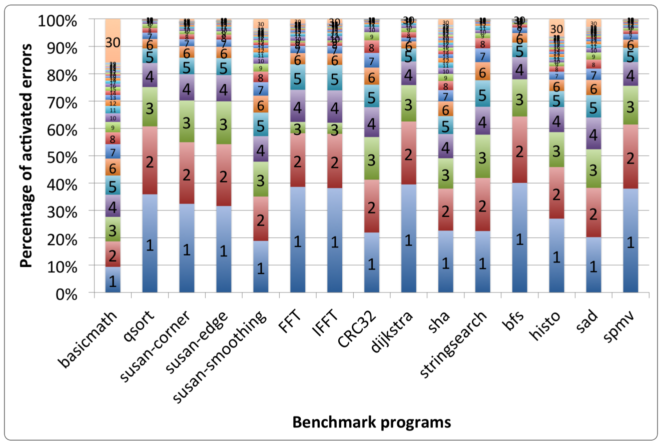
size configurations. In other words, when studying the impact of multiple bit-flip errors on programs, the win-size parameter does not matter much for the SDC percentages. However, Table III shows the win-size configurations that caused the highest percentage of SDCs, among all multiple-bit error campaigns. We can see that when using the inject-on-read technique, *higher window sizes are more likely to result in the highest percentage of SDCs*. This is because a high percentage of data-items targeted by errors when using the inject-on-read technique are memory addresses. Injecting errors into memory addresses are mostly detected by the exception mechanisms (see Fig. 1a). Thus, multiple injections into registers that are within a small window are more likely to result in an address corruption that raises an exception, thereby resulting in a higher percentage of Detections than when consecutive errors are injected into registers that are within a larger window.

Result summary (inject-on-read technique):
***RQ2-Answer*:** The single bit-flip model provides a pessimistic upper-bound on SDCs for most of the programs.
***RQ3-Answer*:** Two errors are enough to cause the highest percentage of SDCs.
***RQ4-Answer*:** Window size does not have much effect on the percentage of SDCs.

b) Results for the inject-on-write Technique: Fig. 5 shows the SDC results for the experiments targeting bits of multiple registers using the inject-on-write technique. The 95% confidence intervals for the results presented here are between ± 0.26 for dijkstra and ± 0.97 for sha. From the figure, we can see that *the single bit-flip model results in a pessimistic estimate of the percentage of SDCs for only around half of the programs*. In the other half, single bit-flip errors result in



(a) inject-on-read



(b) inject-on-write

Figure 3. Distribution of the number of activated errors before causing a program to crash, given that max-MBF is equal to 30.

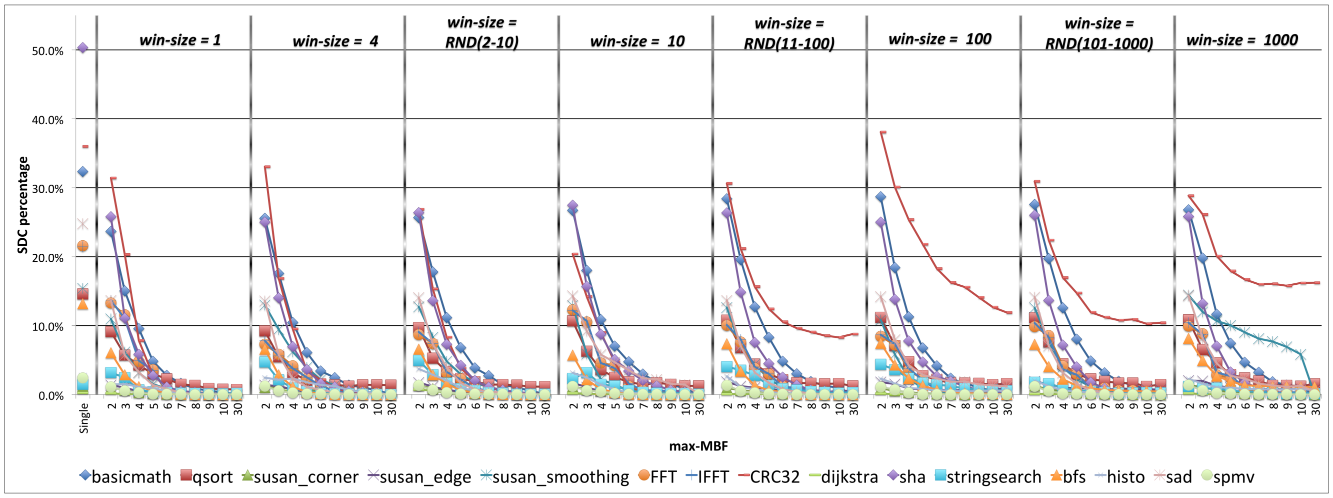


Figure 4. SDC results for experiments targeting bits (from 1 to 30) of multiple registers using the inject-on-read technique. Here the RND (α, β) refers to a randomly selected value between α and β.

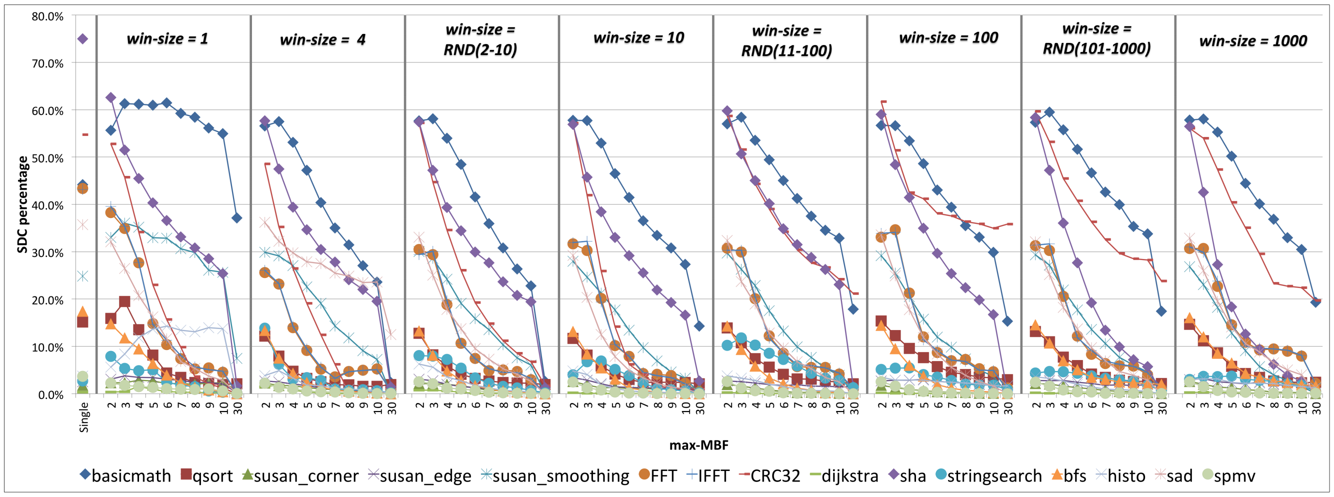


Figure 5. SDC results for experiments targeting bits (from 1 to 30) of multiple registers using the inject-on-write technique. Here the RND (α, β) refers to a randomly selected value between α and β.

2 (for dijkstra) to 17 (for basicmath) percentage points of lower SDCs compared to the multiple bit-flip configurations that cause the highest percentage of SDCs. The high percentage of difference for basicmath could yet again be explained using the results presented in Fig. 1, where injecting single bit-flip errors in basicmath result in the lowest percentage of Detections. This implies that there are fewer hardware exceptions raised in these programs, which means fewer errors are detected and hence results in higher percentage of SDCs.

The results of our multiple bit-flip campaigns show that, in the case of using the multiple bit-flips, *three errors are sufficient to cause the highest percentage of SDCs for 114 out of 120 program/win-size pairs* (corresponding to 95% of the pairs). Out of these 114 pairs, 93 and 21 of them correspond to when the max-MBF is equal to two and three, respectively. Out of the 120 program/win-size pairs, there are also five cases where four errors are needed to cause the highest percentage of SDCs; however, compared to when three errors are injected, these cases only result in at most one percentage point of higher percentage of SDCs, which is not a significant difference. The only exception is the histo program using the window size of one, where six errors are needed to cause the highest percentage of SDCs.

Comparing Fig. 4 and Fig. 5 suggests that depending on the fault injection technique used, different numbers of errors need to be injected into the system to produce a pessimistic estimate of the percentage of SDCs. *However, aggregating the results from both techniques, injecting three errors is sufficient to result in the highest percentage of SDCs.*

Fig. 5 also shows that for many of the programs, the win-size parameter has a significant effect on the percentage of SDCs when using the inject-on-write technique (unlike the inject-on-read technique). Further, according to Table III, *when using the inject-on-write technique, lower window sizes are more likely to result in the highest percentage of SDCs*, which is different from what we observed for the inject-on-read technique. This is because a higher percentage of data-items targeted by errors are data variables, in contrast to the inject-on-read technique where they were address variables. Injecting errors into data variables mostly result in Benign or SDC outcome categories. Thus, by injecting multiple errors within a small window size, the likelihood of causing an SDC increases as there is less opportunity for the effect of an error to be masked before the next injection; thus, we can choose smaller window sizes for pruning.

Result summary (inject-on-write technique):
***RQ2-Answer*:** The single bit-flip model does not result in pessimistic percentage of SDCs for half of the programs.
***RQ3-Answer*:** Three errors are enough to cause the highest percentage of SDCs in 95% of the program/win-size pairs.
***RQ4-Answer*:** Lower window size values are more likely to result in the highest percentage of SDCs.

3) Sensitivity of Fault Injection Locations to Multiple Bit-Flip Errors: In this section, we study whether the results obtained for single bit-flip campaigns can be used to prune the multiple bit-flip error space with respect to the program location into which the first error should be injected. Our goal is to inject the first error of each multiple bit-flip experiment in locations that would eventually cause an SDC and are not covered by the single bit-flip model. We explain this using Fig. 6, which shows how injecting multiple bit-flip errors would affect the results of a single bit-flip experiment.

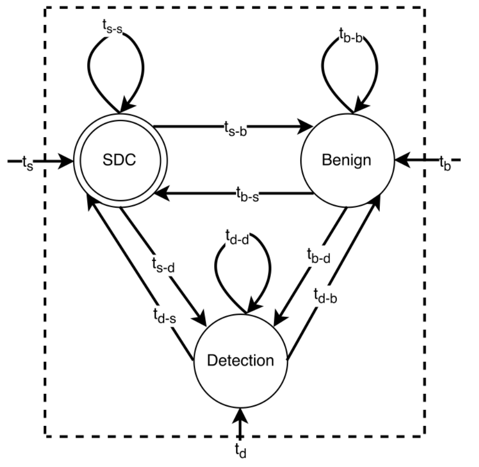


Figure 6. State diagram showing transitions between different outcome categories due to the injection of multiple-bit errors.

In Fig. 6, t_s, t_b and t_d correspond to when a single bit-flip error is injected into a specific location in the program, resulting in SDC, Benign or Detection, respectively. All other transitions correspond to when multiple bit-flip errors are injected starting from the same program location, and change the result of the single bit-flip outcome. For example, t_{b-s} refers to a change in the fault injection result from Benign to SDC due to the injection of multiple bit-flip errors. Fig. 6 illustrates two transitions where injecting additional bit-flip errors into the program changes its result from Benign or Detection to an SDC, thereby decreasing its resilience:

- **Transition I (t_{d-s}).** Injecting single bit-flip error into a location results in the Detection category, but injecting multiple bit-flip errors changes the result to an SDC.
- **Transition II (t_{b-s}).** Injecting single bit-flip error into a location results in the Benign category, but injecting multiple bit-flip errors changes the results to an SDC.

To find the likelihood of the above transitions, we conduct two fault injection campaigns for each program, one for each fault injection technique. To get the worst-case estimates, we use the max-MBF/win-size pairs that caused the highest percentage of SDCs when conducting multiple bit-flip fault injection campaigns (see Table III). We choose the location of the first error of each multiple bit-flip experiment from those chosen for the single bit-flip model. We do not consider the t_{s-s} transition as *we only consider cases that would add to the number of SDCs (i.e., pessimistic percentage of SDCs).*

Table IV shows the results. From the table, we can see that Transition I is very unlikely (in most cases below 1%),

Table IV
LIKELIHOOD OF TRANSITION I AND TRANSITION II.

Program	**inject-on-read**		**inject-on-write**	
	Tran. I	**Tran. II**	**Tran. I**	**Tran. II**
basicmath	1.1%	31.9%	0.6%	58.3%
qsort	0.7%	13.4%	0.4%	29.5%
susan_corner	0.1%	1.2%	0.1%	4.1%
susan_edge	1.2%	0.6%	0.9%	0.8%
susan_smoothing	0.3%	14.6%	0.8%	34.6%
FFT	0.4%	25.6%	4.1%	23.4%
IFFT	0.5%	23.6%	3.6%	26.0%
CRC32	0.8%	48.1%	0.6%	81.8%
dijkstra	0.0%	3.1%	0.2%	2.9%
sha	1.0%	0.0%	2.2%	0.0%
stringsearch	0.1%	7.7%	0.2%	15.7%
bfs	0.2%	10.7%	0.8%	19.2%
histo	0.1%	5.2%	0.2%	19.6%
sad	12.9%	2.9%	14.9%	2.1%
spmv	0.1%	1.1%	0.1%	1.5%

especially with the inject-on-read technique. Therefore, we can prune the multiple bit-flip error space by excluding those locations that would result in the Detection category or an SDC under the single bit-flip model. In fact according to the results presented in Fig. 1, these locations include around 50-100% of the inject-on-read and 27-100% of the inject-on-write single bit-flip experiments, which is a significant reduction in the error space. However, there is much more variation when it comes to the likelihood of Transition II, and its value ranges from 0% to 81%, and hence these locations cannot be ignored.

***RQ5-Answer*:** We can prune the multiple bit-flip error space by injecting the first error of each experiment only into locations that if targeted by a single bit-flip error they would result in Benign outcomes, as these are the locations that would add to the number of SDCs under multiple bit-flips.

V. SUMMARY AND IMPLICATIONS

Our goal was to study the impact of multiple-bit errors in programs and to find ways to reduce the multiple-bit fault injection space (error space). This is important as previous studies [4], [5], [6] have shown that soft errors often manifest as multiple-bit errors at the software level, and hence we need efficient methods to inject multiple-bit errors in software and evaluate their effects. Prior work had considered at most two bit-flips, and did not cover the entire space of multiple-bit errors. We performed a comprehensive analysis of the parameter space of multiple bit-flips to identify which parameters affect the SDCs for a program. Our findings are:

- The SDC results of the single bit-flip model are close to the results for the multiple bit-flip model (except for 2% of multiple bit-flip campaigns which result in more than 5 percentage points of higher percentage of SDCs) across the majority of programs and parameter values, with a few exceptions. This holds regardless of whether the multiple-bit injections are in the same register or in different registers.
- With that said, the single bit-flip model is not sufficient to establish conservative upper bounds on the SDC results (i.e., pessimistic percentages of SDCs) under multiple-bit errors. However, for most programs, the pessimistic percentage of SDCs for the multiple-bit error model is achieved under relatively few multiple-bit errors (2 errors with the inject-on-read technique, and 3 errors with the inject-on-write technique).
- The dynamic window size parameter value does not matter much when the inject-on-read technique is used, but it matters when the inject-on-write technique is used. In the latter case, the highest percentage of SDCs is achieved when the window size is low, i.e., less than 5 dynamic instructions in most cases.
- Only a very small fraction of single bit-flip errors that result in Detection lead to SDCs under multiple bit-flips in which the starting location is the same as the single bit-flip error. Therefore, to maximize the SDCs uncovered by multiple bit-flip injections, one needs to inject only into the program locations in which single bit-flip error injections led to benign outcomes.

Taken together, these results suggest that the *multiple bit-flip error space can be considerably pruned if one is interested in obtaining conservative upper-bounds on SDCs*. In fact, in many cases, the single bit-flip fault injection results already give reasonably close SDC results to the multiple bit-flip injection results. If more accuracy is needed, then injecting a small number of multiple bit-flip errors (at most 3) is sufficient. Furthermore, the multiple fault injections need to be only a few (dynamic) instructions apart, to get conservative upper bounds on the percentage of SDCs. This further helps to prune the error space. Finally, we can leverage the results from the single bit-flip fault injections to choose the locations for multiple bit-flip injections to get conservative SDC results.

In summary, we can conclude that multiple bit-flip errors do not cause as much difference in the SDC results of experiments conducted using single bit-flip errors as some researchers have speculated [4]. Therefore, the single bit-flip fault model may be sufficient for evaluating the coverage of error resilience techniques in most cases. If more accuracy is desired, we need to only consider a limited range of multiple bit-flip errors, which lead to only a modest increase in the error space.

Take-away: The single bit-flip model continues to be a valid approximation for resilience studies, albeit with the above caveats, and hence *one bit is often enough.*

As future work, we plan to extend this study to multiple-bit faults in memory (that are not detected by ECC), as well as consider larger applications. Another potential direction is to consider specific fault tolerance techniques, and measure their coverage with the single and multiple-bit fault models.

Acknowledgment

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC), the Canada Foundation for Innovation (CFI), the EU funded HiPEAC (High Performance and Embedded Architectures and Compilers) Network of Excellence and the Ericsson Research Foundation. We thank Risat Mahmud Pathan, the members of the Dependable Systems Lab at UBC, and the anonymous reviewers of the DSN 2017 conference for their comments, which have helped us to improve this paper.

References

[1] S. Borkar, "Designing reliable systems from unreliable components: the challenges of transistor variability and degradation," *IEEE Micro*, vol. 25, no. 6, pp. 10–16, 2005.

[2] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "SWIFT: Software implemented fault tolerance," in *Proceedings of the 3rd International Symposium on Code Generation and Optimization*, ser. CGO '05. IEEE Computer Society, 2005, pp. 243–254.

[3] M. Rebaudengo, M. S. Reorda, and M. Violante, "A new approach to software-implemented fault tolerance," *Journal of Electronic Testing*, vol. 20, no. 4, pp. 433–437, 2004.

[4] H. Cho, S. Mirkhani, C.-Y. Cher, J. A. Abraham, and S. Mitra, "Quantitative evaluation of soft error injection techniques for robust system design," in *Proceedings of the 50th ACM/EDAC/IEEE Design Automation Conference*, ser. DAC '13. ACM, 2013, pp. 1–10.

[5] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham, "EMAX: An automatic extractor of high-level error models," in *Proceedings of the 9th AIAA Computing in Aerospace Conference*, 1993, pp. 1297–1306.

[6] J. F. Ziegler et al., "IBM experiments in soft fails in computer electronics (1978-1994)," *IBM Journal of Research and Development*, vol. 40, no. 1, pp. 3–18, 1996.

[7] W. G. Bouricius, W. C. Carter, and P. R. Schneider, "Reliability modeling techniques for self-repairing computer systems," in *Proceedings of the 24th National Conference*, ser. ACM '69. ACM, 1969, pp. 295–309.

[8] T. F. Arnold, "The concept of coverage and its effect on the reliability model of a repairable system," *IEEE Transactions on Computers*, vol. C-22, no. 3, pp. 251–254, 1973.

[9] B. Fang, Q. Lu, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi, "ePVF: An enhanced program vulnerability factor methodology for cross-layer resilience analysis," in *46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2016, pp. 168–179.

[10] Q. Lu, M. Farahani, J. Wei, A. Thomas, and K. Pattabiraman, "LLFI: An intermediate code-level fault injection tool for hardware faults," in *2015 IEEE International Conference on Software Quality, Reliability and Security*, 2015, pp. 11–16.

[11] F. Ayatolahi, B. Sangchoolie, R. Johansson, and J. Karlsson, "A study of the impact of single bit-flip and double bit-flip errors on program execution," in *Proceedings of the 32nd International Conference on Computer Safety, Reliability, and Security*, ser. SAFECOMP 2013. Springer-Verlag New York, Inc., 2013, pp. 265–276.

[12] F. Adamu-Fika and A. Jhumka, "An investigation of the impact of double bit-flip error variants on program execution," in *Proceedings of the 15th International Conference on Algorithms and Architectures for Parallel Processing*. Springer International Publishing, 2015, pp. 799–813.

[13] E. Touloupis, J. A. F. Member, V. A. Chouliaras, and D. D. Ward, "Study of the effects of SEU-induced faults on a pipeline protected microprocessor," *IEEE Transactions on Computers*, vol. 56, no. 12, pp. 1585–1596, 2007.

[14] S. K. S. Hari, S. V. Adve, H. Naeimi, and P. Ramachandran, "Relyzer: Exploiting application-level fault equivalence to analyze application resiliency to transient faults," in *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII. ACM, 2012, pp. 123–134.

[15] R. Venkatagiri, A. Mahmoud, S. K. S. Hari, and S. V. Adve, "Approxilyzer: Towards a systematic framework for instruction-level approximate computing and its application to hardware resiliency," in *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–14.

[16] R. Barbosa, J. Vinter, P. Folkesson, and J. Karlsson, "Assembly-level pre-injection analysis for improving fault injection efficiency," in *Proceedings of the 5th European Dependable Computing Conference*. Springer Berlin Heidelberg, 2005, pp. 246–262.

[17] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization*, ser. CGO '04. IEEE Computer Society, 2004, pp. 75–86.

[18] W. Gu, Z. Kalbarczyk, and R. K. Iyer, "Error sensitivity of the linux kernel executing on PowerPC G4 and Pentium 4 processors," in *2004 IEEE/IFIP International Conference on Dependable Systems and Networks*, 2004, pp. 887–896.

[19] D. Skarin, R. Barbosa, and J. Karlsson, "GOOFI-2: A tool for experimental dependability assessment," in *2010 IEEE/IFIP International Conference on Dependable Systems Networks*, 2010, pp. 557–562.

[20] M. de Kruijf, S. Nomura, and K. Sankaralingam, "Relax: An architectural framework for software recovery of hardware faults," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10. ACM, 2010, pp. 497–508.

[21] S. Feng, S. Gupta, A. Ansari, and S. Mahlke, "Shoestring: Probabilistic soft error reliability on the cheap," in *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XV. ACM, 2010, pp. 385–396.

[22] D. S. Khudia and S. Mahlke, "Harnessing soft computations for low-budget fault tolerance," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-47. IEEE Computer Society, 2014, pp. 319–330.

[23] B. Sangchoolie, F. Ayatolahi, R. Johansson, and J. Karlsson, "A comparison of inject-on-read and inject-on-write in ISA-level fault injection," in *11th European Dependable Computing Conference*, 2015, pp. 178–189.

[24] A. Thomas and K. Pattabiraman, "LLFI: An intermediate code level fault injector for soft computing applications," in *Workshop on Silicon Errors in Logic System Effects (SELSE)*, 2013.

[25] J. Pan, "The dimensionality of failures - a fault model for characterizing software robustness," in *Proceedings of the International Symposium on Fault-Tolerant Computing*, 1999.

[26] J. Pan, P. Koopman, and D. Siewiorek, "A dimensionality model approach to testing and improving software robustness," in *Proceedings of the 1999 IEEE AUTOTESTCON*, 1999, pp. 493–501.

[27] L. Rashid, K. Pattabiraman, and S. Gopalakrishnan, "Characterizing the impact of intermittent hardware faults on programs," *IEEE Transactions on Reliability*, vol. 64, no. 1, pp. 297–310, 2015.

[28] H. Madeira and J. G. Silva, "Experimental evaluation of the fail-silent behavior in computers without error masking," in *Proceedings of the 24th IEEE International Symposium on Fault-Tolerant Computing*, 1994, pp. 350–359.

[29] P. Yuste, J. C. Ruiz, L. Lemus, and P. Gil, "Non-intrusive software-implemented fault injection in embedded systems," in *Proceedings of the 1st Latin-American Symposium on Dependable Computing*. Springer Berlin Heidelberg, 2003, pp. 23–38.

[30] H. Schirmeier, M. Hoffmann, R. Kapitza, D. Lohmann, and O. Spinczyk, "Fail*: Towards a versatile fault-injection experiment framework," in *ARCS Workshops*, 2012, pp. 1–5.

[31] H. Schirmeier, C. Borchert, and O. Spinczyk, "Avoiding pitfalls in fault-injection based comparison of program susceptibility to soft errors," in *45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2015, pp. 319–330.

[32] P. Racunas, K. Constantinides, S. Manne, and S. S. Mukherjee, "Perturbation-based fault screening," in *Proceedings of the 13th IEEE International Symposium on High Performance Computer Architecture*, ser. HPCA '07. IEEE Computer Society, 2007, pp. 169–180.

[33] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proceedings of the 4th Annual IEEE International Workshop on Workload Characterization*, ser. WWC-4, 2001, pp. 3–14.

[34] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center for Reliable and High-Performance Computing*, vol. 127, 2012.