An Investigation of the Impact of Double Single Bit-Flip Errors on Program Executions

Fatimah Adamu-Fika and Arshad Jhumka

Department of Computer Science
University of Warwick
Coventry, CV4 7AL UK
Email: {fatimah,arshad}@dcs.warwick.ac.uk

***Abstract*—This paper investigates a novel variant of the *double bit errors* fault model and studies its impact on program execution. Current works have addressed the problem of both random bit upsets occurring in the *same* location (a given memory word or register). In contrast, we randomly *select two locations and flip a single bit at each location*, which we call *Double Single Bit-flip* (DSB) variant. We then evaluate the viability of this new variant in uncovering vulnerabilities in software (SW). As a baseline for comparison, we inject traditional single bit-flip (SBF) errors in registers. To better understand the impact of the injected faults on SW, we classify the behaviour of the program in five possible failure categories. Our results, based on nearly a million fault-injection experiments, show that (i) DSB causes a significantly higher proportion of SW failures than SBF errors, (ii) a large proportion of those failures was crash failure and (iii) under DSB, the proportion of silent data corruptions (SDC) varies significantly between programs from different application areas. The failure profile induced by DSB is very different to other fault models, such as SBF.**

Keywords–Multiple bit-flip errors; Fault injection; Failure profile; Evaluation.

I. INTRODUCTION

With the ever-decreasing size of hardware and issues such as temperature hotspots [1], computer systems are being subjected to increasing rate of transient faults. These transient faults originate from the transistor level. These faults typically cause a corruption of the state of the program, i.e., errors exist in the program [2]. To mimic these errors, bit-flip errors are typically artificially injected into the program state during a process called fault injection [3]. Traditionally, a single bit-flip error was injected in a single run of the program. This involves selecting a variable at a given location in the program and, when execution reaches this location, a single bit upset (SBUs) is performed on the selected variable. However, the increasing rate of transient faults have limited the usefulness of SBUs in uncovering vulnerabilities, necessitating multiple-bit upsets (MBUs) to be injected in a single run.

Fault injection is a widely used technique for the validation of dependable systems. Its importance is being increasingly recognised, with its recommendation as a highly valuable assessment method in the recently published ISO 26262 standard [4] for functional safety of road vehicles supporting this increasing importance. It is expected that single event upsets will likely create MBUs in forthcoming hardware circuits [5][6], including those in embedded systems. In anticipation of this problem, several work have started investigated double-bit upsets (DBUs) fault model [7][8]. However, these works focused on one variant of DBUs: at a given location, *two bits* are randomly selected and are subsequently inverted. There is a rareness of field data on how these hardware errors will manifest. This is also observed in [7][8]. In [9], it has been shown that multiple memory errors may occur as: (i) several bit upsets within a single location, (ii) one or more bit upsets across several locations or (iii) several bits upsets all across the chip. In this paper, we investigate a variant of DBUs: *two locations are selected and a SBU is injected at each location*. We call this new fault model the Double Single Bit-flip (DSB) fault model.

The usefulness of a fault model is its ability to uncover vulnerabilities in a system. Specifically, it is often the case that the *error sensitivity* of a software system is assessed with respect to the errors being injected according to the proposed fault model. Error sensitivity is commonly defined as the likelihood that a softare component will produce a SDC, which is a type of problem that often goes undetected by the system, as a result of a hardware error. It also often the case that the failure profile of the system is evaluated with respect to the fault model. Hence, we have conducted an extensive fault injection campaign, with close to one million fault injection experiments on five different software modules, each with different software structures, to validate the DSB fault model. Our contributions are: (i) we investigate the impact of DSB on program execution, (ii) we conduct a large-scale fault injection experiments, of close to a million executions, to assess the usefulness of DSB, and (iii) our results show that DSB induces very different failure profiles in software than existing fault models, such as SBF. We conclude that DSB is indeed useful in uncovering vulnerabilities.

The remainder of the paper is structured as follows: In Section II, we present the system and fault models we assume in the rest of the paper. We detail the experimental setup used in Section III. In Section IV, we present the results of our experiments. We present an overview of related work in Section V. We conclude the paper in Section VI.

II. MODELS

In this section, we present the system model and the fault model we assume in the rest of the paper.

 ISBN: 978-1-61208-429-9

A. System Model

In this paper, we consider modular software, i.e., software that consists of a number of discrete software functions, called modules, that interact to deliver the requisite functionality. We consider a module as a generalised white-box, having multiple inputs and outputs and whose codebase is available. We do not assume knowledge of the implementation details. The codebase is needed only to enable the software to be instrumented to enable errors to be injected.

Modules communicate with each other in some specified way using different forms of signalling, e.g., shared memory, parameter passing etc. This is usually down to the nature of the software and to the chosen communication model. A software module performs computations using the inputs received on its input channels to generate the outputs, which are then placed on the requisite output channels.

B. Fault Model

Our fault model is transient hardware faults that ultimately affect the software modules. These faults typically originate at the transistor level due to issues such as hardware size and temperature hotspots. These faults affect the state of the program by changing the content of memory and registers (i.e., different locations), causing *errors* [2] to exist in the software. These errors in software are typically mimicked by injecting bit-flip errors in main memory words and registers. In this paper, we focus only on errors in registers and the total number of errors that can occur in any run is two, i.e,, we randomly select two registers and flip one bit in each. We specifically corrupt the contents of registers immediately before they are written into main memory.

III. Fault-Injection Experiments

In this section, we empirically study how DSB and DBF affect program executions. In section III-A, we describe the target programs, the modules that are instrumented in each target program and the input sets that are processed by the programs during injection. We then describe how the modules are instrumented and how the fault injection experiments are done in III-B.

A. Target Programs

We select five different modules from two different software systems for instrumentation. The first system is an image recognition package, SUSAN (Smallest Univalue Segment Assimilating Nucleus) [10], developed for noise filtering and for recognising corners and edges in Magnetic Resonance Image (MRI) of the brain. The second software system is the Mathwork's implementation of a flight control system for the longitudinal motion of an aircraft [11]. We target five different modules within these systems, three from SUSAN and two from the flight control systems.

The three different modules we use in SUSAN are for corners detection, edges detection and noise filtering, which we refer to as corners, edges and smoothing, respectively, in the rest of the paper. We select two modules within the flight control system, (i) the module for updating derivatives for the root system and (ii) the module for updating model step. We refer to these modules as derivatives and step, respectively. Details are provided in Table I for description of input set.

TABLE I. SIZES OF TARGET MODULES AND DESCRIPTION OF THEIR INPUT SET.

Module	Size (bytes)	Input description
Corners	7975	PGM files:
Edges	6053	A simple four-sided geometric shape (7292 bytes)
Smoothing	3488	Multiple geometric shapes of various shapes and sizes (65551 bytes)
		An image (111666 bytes)
Derivatives	2915	Pilot Frequency in rads/secs:
Step	10249	Variable of type unsigned long long
		between 0.030000000000000000 to 0.1199999999999999

B. Experimental Setup

In this section, we provide details about the experimental setup and the fault injection experiments that we conducted.

1) System platform: The experiments were executed on a 3 GHz Intel Core i7 machine, with 16 GB, 1600 MHz DDR3 and 500 GB solid state drive. The machine was running Darwin OS version 14.0.0.

2) Target system: For our faullt injection experiments, we used a variant of LLVM fault injection tool (LLFI) [12], which we refer to as Fault-Rate LLFI (or FR-LLFI) [13]. LLFI works at the LLVM [14] compiler's intermediate representation (IR) level. FR-LLFI allows the injection of faults using a fixed probability, which is called *fault rate*, rather than a single fault per execution. We extended FR-LLFI to allow for multiple bit flips in specific points, we also added the functionality of allowing the selection of what bit(s) to flip at specific points.

To perform a fault injection, we first compiled the source files into a single IR file. The compiled IR file together with a fault injection configuration script (written in PyYaml format [15]) are then fed to the *extended FR-LLFI instrumentor (instrumentor)* for instrumentation. The instrumentor outputs executables (IR and C/C++ object files) to be passed to the *extended FR-LLFI Profiler (profiler)* for profiling and *the extended FR-LLFI fault-injector (fault-injector)* for fault injection.

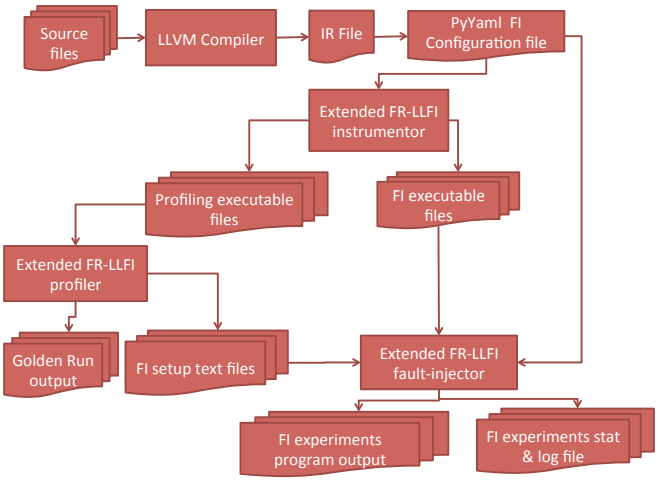


Figure 1. Extended FR-LLFI fault-injection (FI) workflow.

We then passed the executables generated for profiling into the *profiler*. The profiler then generates the setup files (text files) to be used by the *fault-injector* for the fault injection phase. In addition, the *profiler* executes a fault-

free execution of program. This fault-free execution is called *golden run*.

Finally, we fed the setup files generated by the *profiler*, the fault injection executables generated by the *instrumentor* and the fault-injection configuration script into the *fault-injector*.

The *fault-injector* selects an instance of the places of interest specified in the fault injection IR file generated by the *instrumentor* and then inject fault into it at runtime (execution of the fault injection C/C++ object file generated by the *instrumentor*). The output of the *fault-injector* is the fault injection experiments, consisting of program output, log and stat files. Figure 1 depicts the workflow of *extended FR-LLFI*.

TABLE II. VARIABLES SELECTED FOR FAULT INJECTIONS IN DIFFERENT BLOCK LOCATIONS.

Target Program	Module	Variable	Size (bits)	Location (Block)	Alias
SUSAN	Corners	x_size	32	early	SC_A
		y_size	32	early	SC_B
		n	32	central	SC_C
		c	8	central	SC_D
		xx	32	late	SC_E
		yy	32	late	SC_F
	Edges	x_size	32	early	SE_A
		y_size	32	early	SE_B
		n	32	central	SE_C
		m	32	central	SE_D
		c	8	central	SE_E
		w	32	late	SE_F
		x	32	late	SE_G
		y	32	late	SE_H
	Smoothing	x_size	32	early	SS_A
		y_size	32	early	SS_B
		n_max	32	early	SS_C
		x	32	central	SS_D
		center	32	central	SS_E
		area	32	late	SS_F
		tmp	32	late	SS_G
Flight Longitudinal Controller	Derivatives	Integrate_CSTATE	64	early	FD_A
		ActuatorModel_STATE	64	early	FD_B
		Integrategdot_CSTATE	64	early	FD_C
		Wgustmodel_CSTATE	64	central	FD_D
		Qgustmodel_CSTATE	64	central	FD_E
		AlphaSendorLowPassFilter_CSTATE	64	central	FD_F
		StickPrefilter	64	late	FD_G
		PitchRateLeadFilter	64	late	FD_H
	Step	Integrate	64	early	FS_A
		ActuatorModel	64	early	FS_B
		Integrateqdot	64	early	FS_C
		Wgustmodel	64	central	FS_D
		Qgustmodel	64	central	FS_E
		PitchRateLeadFilter	64	central	FS_F
		Gain3_h	64	late	FS_G
		Sum2_g	64	late	FS_H
		Sum1_m	64	late	FS_I

3) Experimental Procedure: To achieve the goals of the study, we run a number of fault injection experiments into a number of different variables (or combinations of variables) in five different modules. We run each target module on three input sets, one from each of three input categories, namely small, medium and large. Before running these experiments, we partition the source code of the program into three parts, namely (i) early, (ii) central and (iii) late. For each part, we choose two or three variables at random, i.e., variables are partitioned and selected according to their placement in the source code of the program. These variables are shown in Table II, with the part of the program source code they belong to. We define a *target location* (or location for short) as a given register used by the program. When a single bit-flip error is injected, a single location is selected. On the other hand, two locations are selected for DSB errors. A fault injection *experiment* is the injection of a an error under the assumed fault model in a given target location. A fault injection *campaign* for a fault model is a set of experiments for a given input set.

Once a location (or pairs of locations) have been selected, we then injected bit-flip errors exhaustively in the locations to cover all possible combination. For each selected location, fault is injected only once during the execution of the program. For the SBU fault model, we ran n experiment in each target location, n being the length of the register. We injected a total *5136* SBUs in the various modules. For the DSB model, for each location pair and a given input, we ran $n \times m$ experiments, m, n being the length of the target locations. Overall, we injected a total of *955392* DSB errors in the software modules. More details can be found in Table II for the size of target locations.

To better understand the profile of the program, we classify the outcome of each fault injection experiment as (i) a *Safe Run*, if the program terminates normally and with an output identical to that of the golden run's, (ii) as a *No Output failure*, if the program terminates normally but fails to produce an output, (iii) as a *Silent Data Corruption (SDC)*, if the program terminates normally but with an output different to that of the golden run's, (iv) as a *Program Hang*, if the program fails to terminate within a predefined time (we set this to 15 times larger than the execution time of the golden run), and (iv) as a *Crash failure*, if the program is terminated due to an exception by the either the program or the operating system.

IV. EXPERIMENTAL RESULTS

We now analyse the results of the various FI experiments, as presented in Tables III – IV and Figures 2 – 4.

A. Impact of DSB vs Impact of Single bit-Flip Error

The first goal of the paper was to evaluate the impact of DSB errors on programs compared to that of single bit-flip (SBF) errors in the same variables. The results for each module are summarised in Table III, while an overall summary is presented in Figure 2.

TABLE III. AVERAGE OUTCOME DISTRIBUTIONS FOR DIFFERENT MODULES.

Module	Fault Model	Outcome				
		Safe Run	*No Output Failure*	*SDC*	*Program Hang*	*Crash Failure*
Corners	SBF	29.4%	18.3%	8.5%	0.0%	43.8%
	DSB	12.5%	13.8%	15.2%	0.1%	58.4%
Edges	SBF	41.5%	0.0%	3.7%	0.0%	54.7%
	DSB	22.0%	0.0%	3.6%	0.1%	74.3%
Smoothing	SBF	14.3%	0.6%	40.0%	0.0%	45.1%
	DSB	6.7%	1.2%	16.4%	10.8%	64.8%
Derivatives	SBF	0.0%	7.1%	0.0%	0.0%	92.9%
	DSB	0.0%	0.5%	0.0%	0.0%	99.5%
Step	SBF	0.0%	25.6%	0.0%	0.0%	74.4%
	DSB	0.0%	6.7%	0.0%	0.0%	93.2%

1) Overall observations: The first observation to be made is that there is marked difference between the failure profile induced by DSB errors compared to that of single-bit flip errors. Further, the proportion of safe run (i.e., no impact) under DSB errors is halved when compared to the proportion of safe runs under SBF errors (see Figure 2). On the other hand, the proportion of crash failure is considerably higher ($\approx 16\%$) under DSB errors than under SBF errors. Also, we observe a reduction in the occurrence of SDCs under DSB errors than under SBF errors. We conjecture that this result is due to the fact DSB errors induce more severe crash

 ISBN: 978-1-61208-429-9

failures, which cause the programs to prematurely exit, and hence such executions cannot display SDCs.

As a matter of contrast, previous work on double bit-flip errors, where two-bit errors are injected into a given location, concluded that single and double bit-flip errors induce very similar proportions of SDCs. We conclude that DSB errors induce a failure profile different to that induced by the double bit-flip errors. As such, we conclude that DSB errors uncover new vulnerabilities in the system and, hence, need to be considered when validating dependable software systems.

2) Module-level observations: From Table III, we observe that the failure profile is dependent on the given target program. For example, we notice the proportion of safe runs under SBF errors in the SUSAN modules is twice as much as that observed under DSB errors. Further, we observe that all faulty runs, irrespective of fault model, in the modules from the control system end in either no output or crash failure. Additionally, we also notice that only the SUSAN modules suffer from SDCs and program hangs under both SBF and DSB errors.

We also observe that the modules from the control system experience mostly crash failures in the presence of DSB errors. Further, we also notice for the control system modules the proportion of no output failure is significantly higher for SBF errors. We also observe a higher proportion of crash failure for DSB errors than that for SBF errors in the SUSAN modules. Given the nature of control systems, which are at the heart of several safety-critical embedded systems, the fact that a high proportion of the DSB errors leads to failures implies that the control systems will not provide reliable service. SDCs have the property that they have not been detected by the system and, thus, provide a potential vulnerability to the system. Also, we observe that DSB errors induce different failure profiles in different modules.

B. Impact of injection location of failure profile

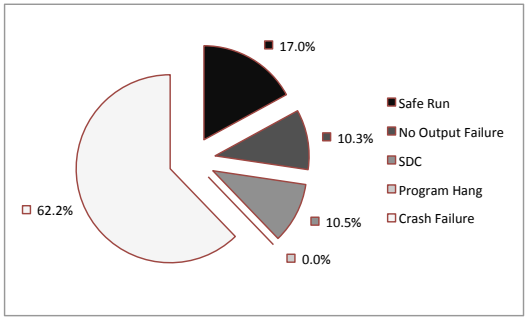
Figures 3 and 4 show the results of the impact injection location has on the failure profile.

1) SBF errors: From Figure 3, the highest proportion, 100%, of safe runs observed in the presence of SBFs is in location SE_D (Figure 3b) and the lowest proportion, 0.0%, is observed in all target locations in derivatives (Figure 3d) and step (Figure 3e).

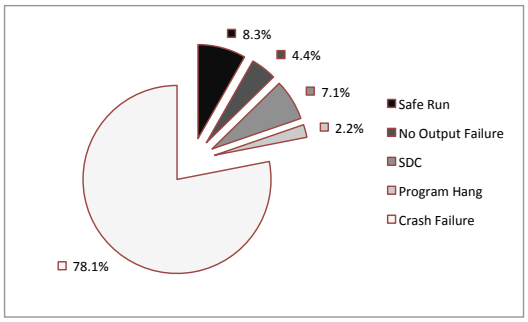
As can be observed from Figure 3, the two modules from the control software suffer a high proportion of crash failures, irrespective of injection location. The other failure type suffered by these two modules are the "no output failure" type. We also observe that, in general, the earlier the injection is performed, the higher the likelihood of a crash failure to happen, i.e., when SBF error is injected in the early part of the modules, the crash failure is more likely to result. On the other hand, crash failure is very likely to happen in the modules of the control software, irrespective of injection location.

2) DSB errors: To understand the impact of injection locations under the DSB errors, we focus on Figure 4 and Table IV

From Figure 4, we observe that failure profiles of the different modules differ from one another. This shows that DSB errors cause these modules to fail differently, thereby



a. Single bit-flip



b. Double single bit-flip

Figure 2. Average outcome distributions over all modules.

inducing different failure profiles in these modules and also that injection locations do affect the failure profiles of these modules.

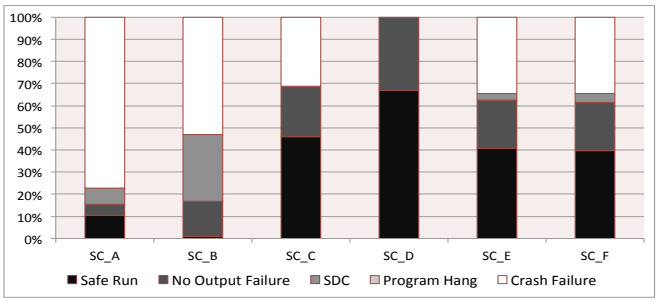
For the two control software modules, any combination of injection locations mostly lead to a crash failure, where the step module suffer a small proportion of the "no output" failure. Focusing on the SUSAN modules, it can be observed that, in general, the earlier an injection is done, the higher the likelihood the failure is a crash failure. On the other hand, it can also be observed that the later an injection is done, the likelihood of a safe run is non-negligible. We now perform a step-by-step comparison between different pairs of injection locations and their respective impact of the software module.

3) DSBs in early blocks vs DSBs in central blocks: We first compare the difference in the impact of DSB errors in early blocks against DSB errors in central blocks, which are shown in Table IV(a) and Table IV(b), respectively. For example, the highest safe run rate observed in early blocks is 0%, whereas the highest safe run rate observed for central blocks is 49.0% (smoothing). The highest proportion of DSB errors in early blocks resulting in crash failure is 99.4% (derivatives) and the lowest is 55.4% (corners), while the highest proportion of SBFs that resulted in crash failure is 99.5% (derivatives) and the lowest, 33.6% (corners).

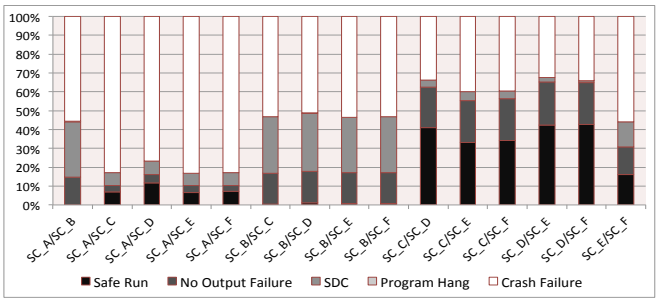
Comparing the results for the different modules, we observe that there is a higher proportion of crash failure, SDCs and program hang when DSB errors are injected in early blocks while, when DSB errors are injected in central block, this results in higher rate of safe run and no output failure.

4) DSBs in early blocks vs DSBs in late blocks: Here, we compare the results of DSB errors in early block with DSB errors in late blocks, as captured in Table IV(a) and

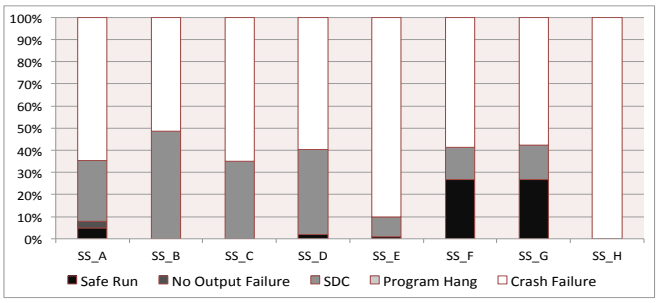
 ISBN: 978-1-61208-429-9



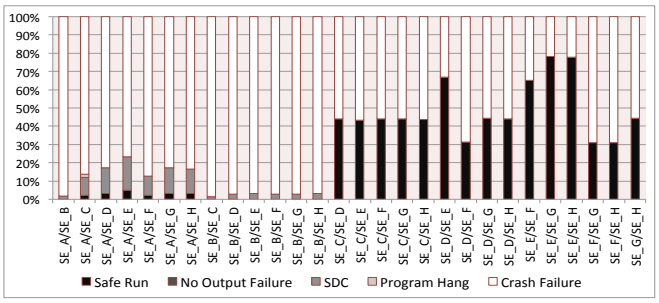
a. Corners



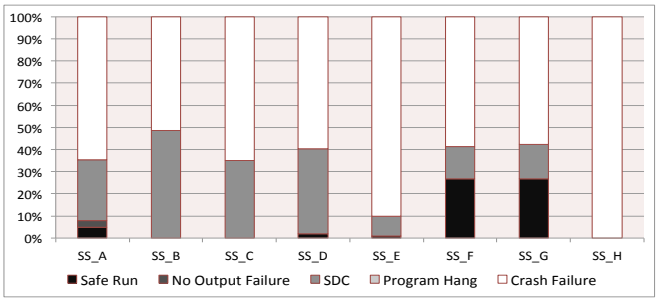
a. Corners



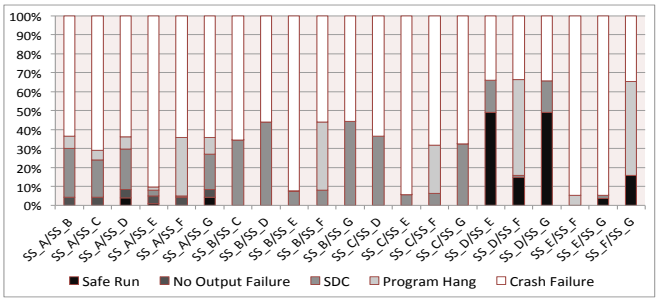
b. Edges



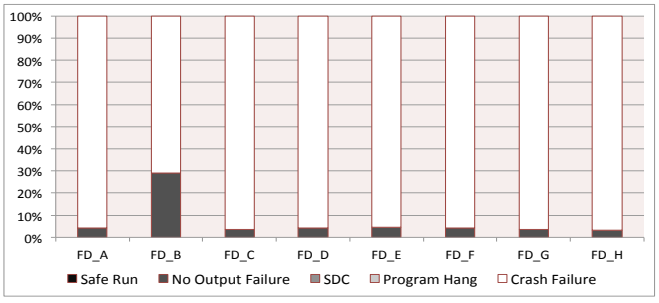
b. Edges



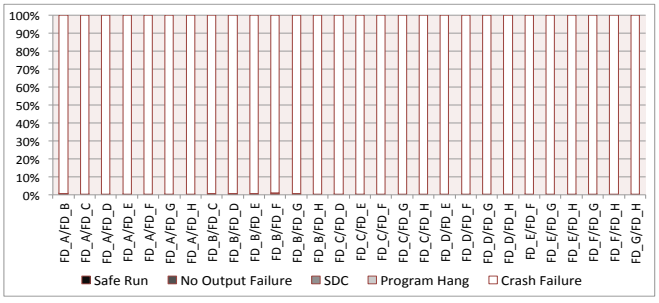
c. Smoothing



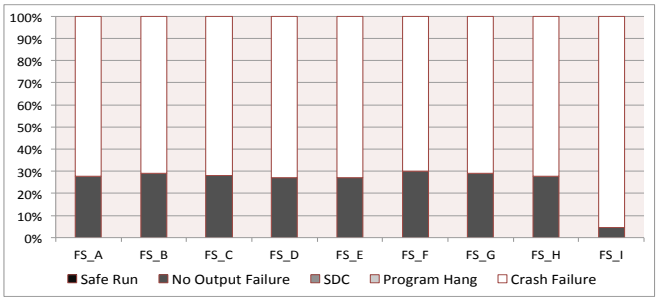
c. Smoothing



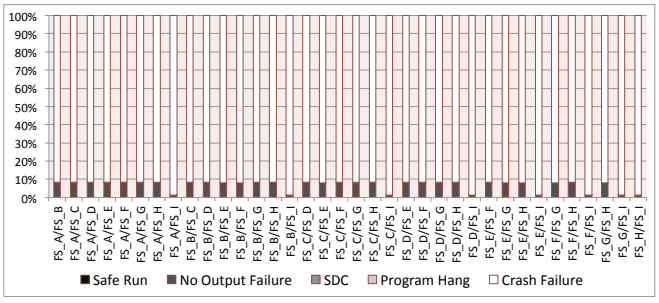
d. Derivatives



d. Derivatives



e. Step



e. Step

Figure 3. Average outcome distributions for SBF experiments for different modules.

Figure 4. Average outcome distributions for DSB experiments for different modules.

TABLE IV. AVERAGE OUTCOME DISTRIBUTIONS FOR DSB IN DIFFERENT BLOCKS COMBINATIONS FOR DIFFERENT MODULES.

(A) EARLY BLOCKS

	Safe Run	*No Output Failure*	*SDC*	*Program Hang*	*Crash Failure*
Corners	0.0%	14.8%	29.3%	0.5%	55.4%
Edges	0.0%	0.0%	1.8%	1.5%	96.6%
Smoothing	0.0%	1.4%	27.1%	4.0%	67.5%
Derivatives	0.0%	0.6%	0.0%	0.0%	99.4%
Step	0.0%	8.3%	0.0%	0.1%	91.7%

(B) CENTRAL BLOCKS

	Safe Run	*No Output Failure*	*SDC*	*Program Hang*	*Crash Failure*
Corners	40.9%	21.7%	3.8%	0.0%	33.6%
Edges	47.8%	0.0%	0.0%	0.0%	52.2%
Smoothing	49.0%	0.0%	16.9%	0.0%	34.1%
Derivatives	0.0%	0.5%	0.0%	0.0%	99.5%
Step	0.0%	8.3%	0.0%	0.0%	91.6%

(C) LATE BLOCKS

	Safe Run	*No Output Failure*	*SDC*	*Program Hang*	*Crash Failure*
Corners	16.1%	14.7%	13.2%	0.0%	56.0%
Edges	35.5%	0.0%	0.0%	0.0%	64.5%
Smoothing	15.7%	0.0%	0.1%	49.7%	34.6%
Derivatives	0.0%	0.4%	0.0%	0.0%	99.6%
Step	0.0%	3.6%	0.0%	0.0%	96.3%

(D) EARLY & CENTRAL BLOCKS

	Safe Run	*No Output Failure*	*SDC*	*Program Hang*	*Crash Failure*
Corners	4.3%	10.0%	18.4%	0.0%	67.3%
Edges	1.5%	0.0%	7.5%	0.3%	90.7%
Smoothing	0.8%	1.4%	19.6%	1.4%	76.7%
Derivatives	0.0%	0.5%	0.0%	0.0%	99.5%
Step	0.0%	8.3%	0.0%	0.0%	91.7%

(E) EARLY & LATE BLOCKS

	Safe Run	*No Output Failure*	*SDC*	*Program Hang*	*Crash Failure*
Corners	3.8%	9.9%	18.1%	0.0%	68.2%
Edges	1.4%	0.0%	7.8%	0.0%	90.8%
Smoothing	0.7%	1.4%	18.4%	16.9%	62.6%
Derivatives	0.0%	0.4%	0.0%	0.0%	99.6%
Step	0.0%	6.0%	0.0%	0.0%	94.0%

(F) CENTRAL & LATE BLOCKS

	Safe Run	*No Output Failure*	*SDC*	*Program Hang*	*Crash Failure*
Corners	35.5%	22.2%	3.8%	0.0%	38.5%
Edges	44.3%	0.0%	0.0%	0.0%	55.7%
Smoothing	16.9%	0.0%	4.9%	14.0%	64.3%
Derivatives	0.0%	0.4%	0.0%	0.0%	99.6%
Step	0.0%	5.9%	0.0%	0.0%	94.1%

Table IV(c), respectively. For example, the highest "no output" failure rate observed in the presence of DSB errors in early blocks in 14.8% (corners) whereas the highest "no output" failure rate observed in the presence of DSBs in late blocks is 14.7% (corners). The highest proportion of DSBs in early blocks resulting in data corruption is 29.3% (corners).

Comparing the results of the different modules, we observe that there is a higher proportion of crash failure, data corruption and no output failure in the presence of DSBs in early blocks, while the presence of DSBs in late blocks result in higher rate of safe runs. Also, the failure profile is more varied (different types of failures) when DSBs are injected in an early block, while the profile is more restricted when DSBs are injected late. Thus, we can conclude that, by *not* injecting in an early block, there is a reduced likelihood of uncovering vulnerabilities.

5) DSBs in early blocks vs DSBs in block combinations: Here, we compare the results of DSBs in early blocks against DSBs in different combinations of blocks, which we present in Table IV(a), Table IV(d), Table IV(e) and Table IV(f), respectively. For example, the lowest crash failure rate seen for DSBs injected in early blocks is 55.4% while the lowest crash failure rate observed for DSBs injected in both early & central block is 67.3%. The highest data corruption rate observed for DSBs in early blocks is 29.3%, while that seen for combination of DSB in early & late blocks is 18.4%. The highest proportion of safe run observed in the presence of DSBs in early block is 0.0%, while the highest observed for DSBs injected in both central & early blocks is 44.3%. Overall, we observed that the proportion of crash failures has increased when DSBs are injected in an early and central block compared when DSBs are injected in an early block only. However, this comes as a counterbalance to a corresponding decrease in SDCs when DSBs are injected in an early and central block.

We also observed that injecting DSBs in an early and central block results in very similar failure profile as when injecting DSBs in an early and late block. On the other hand, we observed that when DSBs are injected in a central and late block, the profile changes considerably. The proportion of safe runs increases while the proportion of crash failures decreases (except for the control software). Thus, with these results, we can conclude that the locations at which DSBs are injected has a strong impact on the failure profile of the system. We have shown that an early injection of a DSB error often leads to a failure.

C. Limitations

One limitation of the results presented here is the range of applications we have used to evaluate the DSB fault model. Though initial results show that the fault model can help uncover vulnerabilities that are otherwise not detected by single bit-flip errors, the fault model needs to be validated against several other applications.

A second limitation in the results presented here is that, to the best of our knowledge, there is little to no field data that shows how multiple bit upsets will manifest themselves. There is however increasing evidence that the rate of hardware errors is increasing. We only consider DSB errors here and, in our future work, we are considering multiple single bit-flip (MSB) errors. The relevance of the results presented here is only as far as the field data matches the DSB model introduced.

V. RELATED WORK

Fault injection is a widely used technique in dependability evaluation [7][12][16][17]. Hardware transient faults are injected into a target system by flipping bits in CPU registers or memory [16][17]. Recent research have shown that multiple fault injections can be very effective in detecting software vulnerabilities [7][8]. Other works have investigated impact of device-level fault injections that manifest as single bit-upsets in registers and main memory [18][19][20].

 ISBN: 978-1-61208-429-9

Recently, the effects of multiple bit-upsets on SRAMs and DRAMs have been studied. In [21], the authors investigated DRAM disturbance errors that manifests as multiple bit-upsets in memory. On the other hand, the authors of [22] investigated the geometric effects of multiple bit-upsets injected into DRAMs. The main difference between our study and these studies is the level of abstraction we focused on. The fault model under investigation in [22] is multiple bit-upsets in multiple cells within the same memory location while that under investigation in [21] is multiple bit-upsets in different memory locations. In spite of the fundamental differences between our work and theirs, they also showed higher rate of safe runs under the single bit-flip model. In addition, under the double bit-flip model, higher crash failure rate is observed. However, they reported that the proportion of SDCs is higher under the double bit-flip model, this is contrary to what our study showed. We observed lower proportion of SDCs under the variant of double bit-flip model studied here than when compared with the single bit-flip model.

Similar to our study, the authors of [8] mimicked bit-flips in registers of a real hardware platform. In addition, they investigated the impact of SBF and double bit-flips (two random bit-flips in same location) on program execution. Our study mainly differs from theirs in the assumed DBU fault model. The DBU fault model in their work selects a single location and flips two bits in that location, while in ours the model chooses two locations and flips one bit in each location. However, in [8], they also injected faults in memory words and investigated the error sensitivity for different target locations. Both works reported a higher level of safe runs for SBUs and a higher proportion of crash failures for DBUs.

VI. Conclusion and Future Work

We have investigated the impact of a novel variant of the double bit upsets, namely the double single bit-flip model, on software execution. We have evaluated it on five different modules from two different applications. Our results show that (i) the proportion of crash failures induced by DSBs is significantly higher than single bit-flip errors, (ii) the proportion of SDCs is lower with DSBs than with single bit-flips and (iii) DSBs induce different failure profiles in different applications.

As future work, we will investigate the reason behind the observed differences between this model and SBF. We will also extend the DSB model to include injection in memory words. Further, we will compare the failure profile of the DSB model with existing DBU models. We will also investigate the effectiveness of current software-based fault tolerance techniques, such as detectors, against DSBs and in turn determine the type of fault tolerance needed to handle the different types of failures. We will also generalise the work focusing on multiple single bit-flip (MSB) errors (instead of two).

References

[1] C. Yang and A. Orailoglu, "Processor reliability enhancement through compiler-directed register file peak temperature reductionprocessor reliability enhancement through compiler-directed register file peak temperature reduction," in Proceedings Dependable Systems and Networks, 2009, pp. 468–477.

[2] J.-C. Laprie, Dependability: Basic Concepts and Terminology. Springer-Verlag, December 1992.

[3] M. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault injection techniques and tools," IEEE Computer, vol. 30, no. 4, April 1997, pp. 75–82.

[4] "Iso 26262-1:2011, road vehicles – functional safety – part 1: Vocabulary. iso, geneva, switzerland," 2011.

[5] G. Georgakos, P. Huber, M. Ostermayr, E. Amirante, and F. Ruckerbauer, "Investigation of increased multi-bit failure rate due to neutron induced seu in advanced embedded srams," in IEEE Symposium on VLSI Circuits, 2007, pp. 80–81.

[6] R. Reed and et al., "Heavy ion and proton-induced single event multiple upset," IEEE Transactions on Nuclear Science, vol. 44, no. 6, 1997, pp. 2224–2229.

[7] S. Winter, M. Tretter, B. Sattler, and N. Suri, "simfi: From single to simultaneous software fault injections," in Proceedings of Dependable Systems and Networks (DSN), 2013.

[8] 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 Computer Safety, Reliability, and Security, ser. Lecture Notes in Computer Science, F. Bitsch, J. Guiochet, and M. Kaniche, Eds. Springer Berlin Heidelberg, 2013, vol. 8153, pp. 265–276. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-40793-2_24

[9] X. Li, M. C. Huang, K. Shen, and L. Chu, "A realistic evaluation of memory hardware errors and software system susceptibility," in Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference, ser. USENIXATC'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 6–6. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855840.1855846

[10] S. Smith, "Susan version 2l," http://users.fmrib.ox.ac.uk/~steve/susan/susan2l.c, 1999, [Online; accessed 19-November-2014].

[11] MATLAB, version 8.3 (R2014a). Natick, Massachusetts: The MathWorks Inc., 2014. [Online]. Available: http://www.mathworks.co.uk/products/matlab/

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

[13] S. Smith, "Fault-rate llfi," https://github.com/ShadenSmith/LLFI, 2014, [Online; accessed 19-November-2014].

[14] 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: Feedback-directed and Runtime Optimization, ser. CGO '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 75–. [Online]. Available: http://dl.acm.org/citation.cfm?id=977395.977673

[15] PyYaml, "Pyyaml," http://pyyaml.org/wiki/PyYAMLDocumentation, 2011, [Online; accessed 19-November-2014].

[16] M. Hiller, A. Jhumka, and N. Suri, "An approach for analysing the propagation of data errors in software," in Proceedings of the 31st IEEE/IFIP International Conference on Dependable Systems and Networks, July 2001, pp. 161–172.

[17] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham, "Ferrari: A flexible software-based fault and error injection system," IEEE Transactions on Computers, vol. 44, no. 2, February 1995, pp. 248–260.

[18] B. Sangchoolie, F. Ayatolahi, R. Johansson, and J. Karlsson, "A study of the impact of bit-flip errors on programs compiled with different optimization levels," in Dependable Computing Conference (EDCC), 2014 Tenth European, May 2014, pp. 146–157.

[19] D. Di Leo, F. Ayatolahi, B. Sangchoolie, J. Karlsson, and R. Johansson, "On the impact of hardware faults — an investigation of the relationship between workload inputs and failure mode distributions," in Proceedings of the 31st International Conference on Computer Safety, Reliability, and Security, ser. SAFECOMP'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 198–209. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-33678-2_17

[20] M. Demertzi, M. Annavaram, and M. Hall, "Analyzing the effects of compiler optimizations on application reliability," in Workload Characterization (IISWC), 2011 IEEE International Symposium on, Nov 2011, pp. 184–193.

 ISBN: 978-1-61208-429-9

[21] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, “Flipping bits in memory without accessing them: An experimental study of dram disturbance errors,” SIGARCH Comput. Archit. News, vol. 42, no. 3, Jun. 2014, pp. 361–372. [Online]. Available: http://doi.acm.org/10.1145/2678373.2665726

[22] S. Satoh, Y. Tosaka, and S. Wender, “Geometric effect of multiple-bit soft errors induced by cosmic ray neutrons on dram’s,” Electron Device Letters, IEEE, vol. 21, no. 6, June 2000, pp. 310–312.

 ISBN: 978-1-61208-429-9