Efficient Soft Error Vulnerability Estimation of Complex Designs

Shahrzad Mirkhani
University of Texas at Austin
shahrzad@cerc.utexas.edu

Subhasish Mitra
Stanford University
subh@stanford.edu

Chen-Yong Cher
IBM TJ Watson Research Center
chenyong@us.ibm.com

Jacob Abraham
University of Texas at Austin
jaa@cerc.utexas.edu

Abstract

Analyzing design vulnerability for soft errors has become a challenging process in large systems with a large number of memory elements. Error injection in a complex system with a sufficiently large sample of error candidates for reasonable accuracy takes a large amount of time. In this paper we describe RAVEN, a statistical method to estimate the outcomes of a system in the presence of soft errors injected into flip-flops, as well as the vulnerability for each memory element. This method takes advantage of fast local simulations for each error injection, and calculates the probabilities for the system outcomes for every possible soft error in a period of time. Experimental results, on an out-of-order processor with SPECINT2000 workloads, show that RAVEN is an order of magnitude faster compared with traditional error injection while maintaining accuracy.

1 Introduction

Analyzing a design for radiation-based soft errors is an involved process, but necessary for life- and mission-critical systems. Results from [16] and [2] show that single bit-flip on the flip-flops of a system are suitable candidates for soft error modeling, since the system outcomes, when these errors are injected, are statistically close to the outcomes of a chip under radiation. Despite the fact that this model is accurate, analyzing a system for all soft error candidates (or even a sufficient sample of single bit-flips) takes a long time and a large amount of computational resources (e.g., processor cores). For a design with thousands of flip-flops, analyzing each flip-flop's vulnerability, for adding resiliency to the design demands even more injections with an acceptable margin of error.

During the past two decades, there have been methodologies developed to improve the run times for error injection of bit-flips on flip-flops. These methods are categorized below. A more detailed categorization and discussion can be found in [14].

- Hierarchical simulation methods that switch between different levels of abstraction, like transistor-level, gate-level, RT-level, and architectural level [3] [5] [15] [17].
- Analytical methods that use static analysis of circuits and probabilistic models of components (e.g., gates) to estimate soft error resilience of systems [1] [6] [7].
- Architectural-based methods that take advantage of architectural features and high-level (instruction-level) simulation passes to estimate the vulnerability [9] [11] or the worst-case soft error rate [12] [18] of the systems.
- System emulation that uses FPGAs to build a prototype of the hardware that runs much faster than simulation [13].

In this paper, we introduce a novel soft error vulnerability estimation called RAVEN (RApid Vulnerability EstimatioN). In RAVEN, we take advantage of statistics from fast local simulations [10] to build a toolflow that can calculate the detection probabilities of soft error candidates in the whole system. In RAVEN, the probability of DUE (Detected Unrecoverable Error) and SDC (Silent Data Corruption) outcomes for all possible soft error candidates in a period of time is calculated. In this paper, we use single bit-flip on each flip-flop as our error model since the result of error injection with this error model is close to radiation-based injections [16][2]. On the other hand, the results of high-level error injection are considerably different from the results of the injection for this error model [4].

We have compared RAVEN run times and outcome probabilities with error injection for a sample of error candidates. We have chosen different sample sizes in our analysis, and for large enough sample sizes (required for an acceptable margin of error), RAVEN runs 2 to 3 orders of magnitude faster than error injection. In our experiments on the IVM processor model [17] with SPECINT2000 workloads, all of the benchmarks lie in the outcome range that is calculated by error injection, except for the workload *crafty*. To investigate further, we have performed complete error injection targeting a small subset of error candidates and 400 cycles in *crafty* and compared the results of flip-flop vulnerability factors[1] that RAVEN calculates to those in complete injection and sampling error injection. These results show the accuracy and speed of RAVEN over sampling error injection when detailed vulnerability is necessary for deciding on resilience techniques for that design.

RAVEN can be used with existing techniques to further improve the efficiency of soft error analysis. For example, RAVEN can be used in FPGA-based systems or hierarchical simulation environments. To show the speed advantage of RAVEN in a proven, efficient simulation environment, we use a hierarchical simulation environment as described in [17]. This environment can accelerate error propagation by switching between an instruction-set simulator and Verilog RTL simulator. All of our simulations (RAVEN and error injection) are done in this hierarchical simulation environment.

[1] $\frac{\#\ of\ erroneous\ outcomes}{\#\ of\ total\ injections}$ for errors injected into a flip-flop

978-3-9815370-4-8/DATE15/©2015 EDAA

In Section 2, we will describe RAVEN methodology. Section 3 describes our experimental results. Section 4 discusses the speed-up of RAVEN over complete error injection and the effectiveness of RAVEN in analyzing designs for resilience.

2 RAVEN Methodology

As mentioned above, RAVEN is a statistical method to estimate the probability of DUE and SDC outcomes for all possible soft error candidates in a period of time. This method works faster than traditional error injection methods. However, using local fault simulations and probabilistic calculations, this method could introduce inaccuracies. An apples to apples comparison of RAVEN with existing error injection techniques would require performing error injection for every possible soft error candidate for long periods of execution. Since this would require inordinate amounts of time, we have performed complete error injection for a short period and a small sub-set of flip-flops and have shown that RAVEN can make more accurate estimates of the flip-flop vulnerability factors (and also average outcomes) than error injection with sampling. In the following sub-sections, we will discuss how RAVEN calculates the outcome probabilities for each soft error candidate.

2.1 RAVEN Steps

RAVEN takes advantage of local simulations which are significantly faster than simulating the entire system. By following the four steps listed below, we can estimate the outcomes for given workloads under soft error candidates in a period of time. We first partition the design, using the design hierarchy as a guide, so that each partition contains one or more modules/entities of the design, depending on their size. For example, in a pipeline-based processor, each pipeline stage can be one partition. Partitioning a design for RAVEN is not limited to the pre-defined modules; any partitioning algorithm can be used in RAVEN. However, the more we avoid system-level feedback loops and system-level reconvergent fanouts, our estimation will be more accurate. Also, we need to avoid very large partitions since they will degrade RAVEN run-time efficiency.

Step 1: Performing one pass of *golden* simulation (with no error injection) for the whole system and storing the values for each partition's inputs/outputs in a file.

Step 2: Building a table (**propagation table**) for each partition *P* which shows the probability of error propagation from each input of *P* to each output of *P*, based on the values from the previous step.

Step 3: Building a table (**detection table**) for each partition *P* which shows the probability of error propagation from each source of error inside *P* to each of *P*'s outputs, based on the values in Step 1.

Step 4: Calculating the detection probability of each soft error candidate in the entire system.

In the following sections, we will describe the above steps in more detail.

2.2 Propagation Tables

Propagation tables show the error propagation probability from each input to each output of a partition. According to this definition, we need to know how many times an error at one of the inputs is observed at any outputs. For this purpose, we inject single stuck-at fault at each input of a partition for each cycle and measure how many times this error is observed at any output. Then, we define the probabilities by dividing the number of error detections at each output, by the number of cycles. We define this average number as the propagation probability of an error on a local input to a local output. For example, in Fig. 1, partition *P* has n inputs and 2 outputs and suppose we are analyzing *P* during 10 clock cycles. To generate *P*'s propagation table, we perform $2 \times n$ simulation passes and inject a stuck-at 1 and a 0 on each input, for each simulation pass. In each simulation pass, we count the number of times that the values of o_1 or o_2 are not the same as their stored golden values. Then we calculate the propagation probability from i_1 to o_1, for instance, by dividing the number of times that o_1 is different than its golden value when i_1 is faulty, by 10 (the number of cycles). Equation 1 shows the propagation probability from input i to output o of a partition. In this equation, *i_sa_0/1* means line i stuck-at-0 or stuck-at-1. We add these two cases to obtain a rough estimate of the probability of an error on the output when there is an error on each input [2].

$$prop_table[i][o] = \frac{\#\ of\ cycles\ i_sa_0/1\ detected\ on\ o}{total\ \#\ of\ cycles} \quad (1)$$

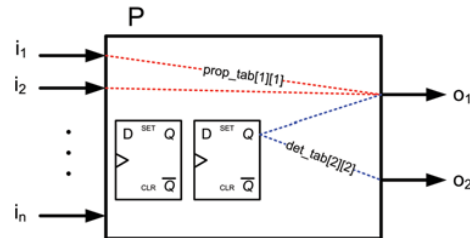

Figure 1: A sample module with n inputs, 2 outputs, and 2 flip-flops.

We should note that we can also use fault simulation tool (instead of injecting the errors into inputs serially) and perform one pass of fault simulation to calculate this table.

2.3 Detection Tables

A detection table for each partition *P* indicates the probability that a soft error in *P* will be seen at one or more outputs of *P*. Similar to propagation table elements, we can calculate each element of detection table by injecting a stuck-at fault into a flip-flop, simulating the partition in presence of that fault, and comparing each output with its golden value at each cycle. Dividing the number of observations of an error on an output by the total number of cycles will give us the probability of detection of that error for that output. However, this number shows the probability of an error detection when that error is present on the flip-flop in all cycles. To model soft errors, we need to assume that a soft error affects the output of a flip-flop in just one (or a few) cycle(s) and therefore, to have a more accurate detection probability, we multiply our original detection probability by $\frac{error\ duration}{\#\ of\ cycles\ error\ can\ be\ present}$ to model a soft error. We assume that the error duration is 1 cycle in this paper. Such an assumption is not necessary in propagation tables, since we

[2] If we assume the direction of the bit-flip ($1 \rightarrow 0$ or $0 \rightarrow 1$) in our probability tables, we will have a more precise estimation. However, this needs 4x more storage than the current rough estimation.

are calculating the average propagation of an error from a local input to a local output. One advantage of RAVEN is that if we change the error duration, we only need to re-calculate the detection probabilities without needing to re-run the local simulations. In Fig. 1, if we want to generate the detection table of partition *P*, we need 4 passes of simulation for flip-flop stuck-at-0 and stuck-at-1 faults. Then we need to count the number of erroneous outputs observed for each pass, and divide them by 10 (number of cycles). To consider these errors as soft errors we should multiply the average we obtained by $\frac{1}{10}$. Equation 2 shows the detection probability for error e to output o. o is an output of a partition and e is a soft error candidate in that partition. Usually, the number of cycles that e can happen is equal to the total number of cycles.

$$det_table[e][o] = \frac{\#\, of\, cycles\, e_sa_0/1\, detected\, on\, o}{total\, \#\, of\, cycles} \times \frac{duration\, of\, e}{\#\, of\, cycles\, e\, can\, happen} \tag{2}$$

2.4 System Level Detection Probability Calculation

After generating the propagation and detection tables for each partition, we calculate the detection probability for each soft error on a flip-flop in the system, as a whole, by using the detection and propagation tables of the partitions in the system. First, to calculate the outcome of an error e at the system level, we pick the detection table elements in partition P corresponding to e. Then we update the output values of partition P due to these elements. Next, we update the outputs of each partition that have inputs connected to the outputs of P by using their propagation probability tables. We continue these calculations until we reach a signal of interest, which will be discussed in Section 2.5. The probability observed on this signal is the detection probability of e in the whole system. We need to perform this process for each flip-flop in which we are interested.

To calculate the detection probability of the outputs of a partition due to its input detection probabilities, we use a simple probability union calculation based on the input probabilities and the corresponding propagation table elements. We use Eq. 3 to calculate the detection probability of an error on output o in partition *P*.

$$det_prob[o] = 1 - \prod_{i=1}^{I_p} (1 - prop_tab_p[i][o] \times det_prob[i]) \tag{3}$$

In Eq. 3, I_p is the number of inputs of partition *P*, $prop_tab_p[i][o]$ is the propagation probability from input *i* to output *o*, and $det_prob[i]$ is the detection probability on *i*.

2.5 Outcome Probability Calculation

In soft error injection, we observe the outcome of a specified workload with an error candidate injected in the design. The outcomes that we discuss in this paper are SDC and DUE outcomes. In the former, the program exits normally, but the output of the program is not correct, or the state of the system is not equal to the golden state. In our experience, if an error stays in the system for a long time it is very probable that it causes an SDC outcome (as seen in more than 93% in our workloads). Therefore, in order to speed up the evaluation, we compare the state of the system after a certain amount of cycles, and if the golden and erroneous states do not match, we assert a dummy output called "sdc". In the latter case (DUE), the program exits prematurely. We check all the conditions that cause a program to halt (e.g., instruction exception) and assert a dummy output, called "halt", under any of these conditions. Since we perform local simulations, we are not able to find out some system-level cases that make the program to run infinitely (i.e., cause a "hang" outcome, for example). Although this outcome happens very rarely, it should be considered as a source of inaccuracy in RAVEN.

In our calculations, if an error causes the pipeline to stall persistently or causes an exception in a clock cycle, this exception will happen at every clock cycle from that cycle until the end of the simulation. Therefore, the propagation probabilities in these cases are skewed to the cycles closer to the end of simulation. In calculating the probability of a "halt" outcome based on the probability of its corresponding partition's inputs, we simply calculate the maximum propagation probability from each input path to the "halt" output instead of calculating the "halt" output probability based on Eq. 3. As a future work, we will consider the skew in every calculation and will improve Eq. 3 to consider the skew as well as the probability values.

When we calculate the "halt" and "sdc" detection probabilities for each flip-flop in RAVEN, we need a way to calculate the outcome probability for DUE and SDC outcomes for a set of error candidates. From our definitions of detection and propagation probabilities, each outcome detection probability for each flip-flop gives a rough estimate of how many times a soft error on a specified flip-flop causes an erroneous outcome during our simulation. Therefore, if we average the detection probabilities on the "sdc" and "halt" dummy outputs over the number of flip-flops, we can have outcome probabilities (an estimation of outcome rate) for all soft error candidates (like Eq. 4 for SDC outcome).

$$sdc_outcome = \frac{\sum_{f=1}^{FF_num} det_prob_f^{sdc_dummy_out}}{FF_num} \tag{4}$$

An advantage of RAVEN is that the detection probability of each error in the whole system can represent the vulnerability factor for its respective flip-flop as well. We can use these vulnerability factors as a guide to choose an efficient resilience technique for our design.

3 Experimental Results

We have implemented RAVEN and applied it to IVM [17], which is an out-of-order and super-scalar Alpha-based processor. In order to partition this processor, we simply make each stage of the pipeline a partition, and put the rest of the "glue" logic in a separate partition. Next, we change the top-level Verilog model, which loads a program into IVM memory and starts the processor, so that it can store the golden input and output values of each partition in a file, as well as the golden state of the processor on the cycle that we need to start our local simulations. These files are generated during one pass of golden simulation (simulation with no error injection). We use these files in the table generation step. For propagation table generation, we have developed a Verilog model for each partition. This model reads the golden inputs/outputs from the files we stored in the golden simulation step, changes one input and simulates the Verilog model of that partition. Then at

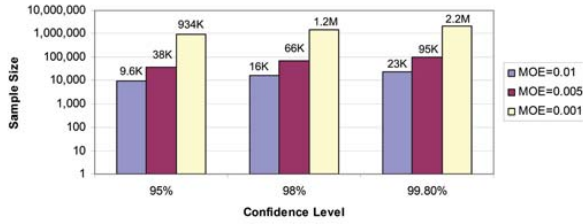

Figure 2: Maximum sample sizes for different absolute MOE values.

each cycle, it compares the output with the golden output, and for the outputs that are different from the golden output, the corresponding table element is updated. After one simulation pass, the system is reset to its initial state and it is ready for the next input error injection. Detection table generation is a similar process; however, instead of input error injection, the error is injected on each flip-flop in that partition.

For our system-level probabilistic calculation, we have developed a SystemVerilog model which simply replaces each *wire/reg* signal type with *real* type so that we can pass probability values instead of logic values. Each partition is replaced by a process, sensitive to all of its inputs. Inside this process is a function implementing Eq. 3. Also, all propagation tables and detection tables are included in this model.

We have estimated the outcome probabilities for some parts of SPECINT2000 workloads, with MinneSpec input data, on IVM. In this paper, we compare RAVEN with statistical fault injection (SFI [8]), which uses samples of error candidates for injection. For SFI, we follow the methodology used in [17], which runs the program in RTL for a limited number of cycles and the rest of the program runs in an instruction set simulator. Error injections are done during RTL simulation. Therefore, we also run RAVEN for this limited simulation period[3].

We have executed RAVEN for IVM on a Dell™ PowerEdge R720 system, with two Intel™ Xeon E5-2690 (2.90 GHz) and 384 GB of random access memory. All error injections on IVM have been executed on Stampede supercomputer. Therefore, to compare RAVEN and SFI run times, we extrapolate the run times for SFI as the number of injections multiplied by the time spent for one good simulation on the Dell™ machine that RAVEN was executed. Note that we have not counted cases for program time-outs and the time for injecting an error. Therefore, our extrapolation is fair for SFI. Note that each benchmark has a different run time, since we run different number of instructions for each benchmark. All numbers for SFI run times follow the extrapolation mentioned above.

In this paper, we compare RAVEN with SFI with different sample sizes. Due to [8], we can calculate the sample size (n) using Eq. 5. In this equation, e is the absolute margin of error (MOE) of sampling. p is the estimated outcome rate that we gain. We usually put 0.5 for p to gain the maximum sample size, since we do not know initially what outcome rates we are going to gain in SFI. N is the set of all error candidates, and t is a cut-off point corresponding to a confidence level. In this paper, we have performed our calculations based on 95%, 98%, and 99.8% confidence levels (with corresponding t=1.96, 2.5758, and 3.0902).

$$n = \frac{N}{1 + e^2 \times \frac{N-1}{t^2 \times p \times (1-p)}} \tag{5}$$

Using Eq. 5, we have calculated the sample sizes for 3 confidence levels and three different MOEs (0.01, 0.005, and 0.001). This is shown in Fig. 2. As can be seen in this figure, the sample size grows significantly when we decrease the MOE. Due to our relatively small outcome rates for DUE and SDC, which are approximately between 2% to 14%, even a small MOE can cause a quite large percent relative MOE in our calculations. For example, if we have SDC rate equal to 4% and our absolute MOE is 0.01 (=1%) it means that the SDC outcome in this case lies between 4%±1%. This means that the percent relative MOE in SDC rate calculation is 25%.

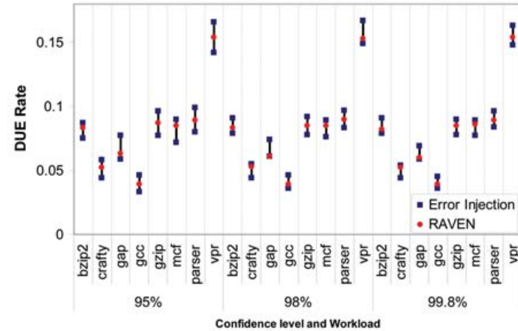

Figure 3: Range of DUE rates in SFI and DUE probability in RAVEN (MOE=0.01).

Based on the sample sizes in Fig. 2, we have performed SFI for three confidence levels and MOE equal to 0.01. The percent relative MOE of the calculated SDC and DUE rates are between 5% to 22%. Therefore, to have small percent relative MOEs, we need to set e to even a smaller number.

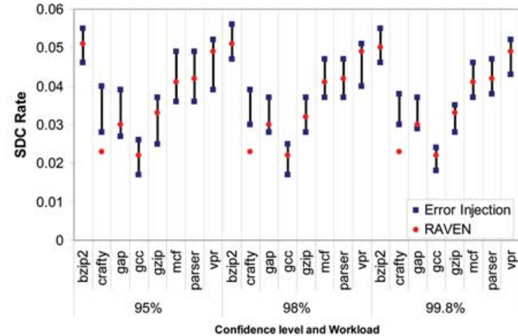

Figure 4: Range of SDC rates in SFI and SDC probability in RAVEN (MOE=0.01).

We have estimated DUE and SDC outcome probabilities for all the error candidates in a time interval for 8 SPECINT2000 workloads on IVM and also performed SFI with 3 different sample sizes (for confidence level=95% and MOE=0.01). In Figures 3 and 4, the outcome rates of RAVEN vs. SFI for these workloads and each sample size is shown. In Fig. 5, we have compared the run times of RAVEN and SFI. RAVEN run times are different for each sample size, since we have calculated the outcome probability for only those flip-flops that were used in

[3]In this paper, the injections are done during 2400 cycles. However, the efficiency of RAVEN becomes even more apparent if we increase the number of cycles in the RTL simulation.

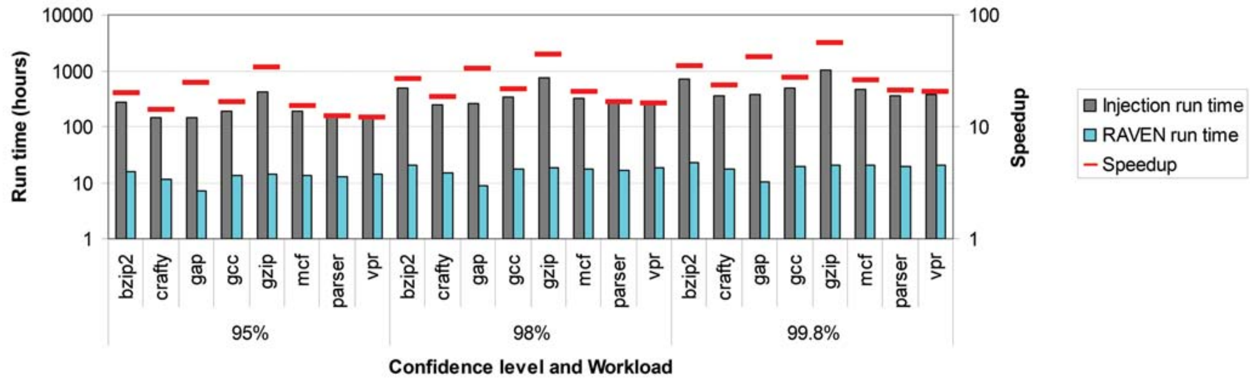

Figure 5: Run times for RAVEN and SFI methods, along with the speed-up (MOE=0.01).

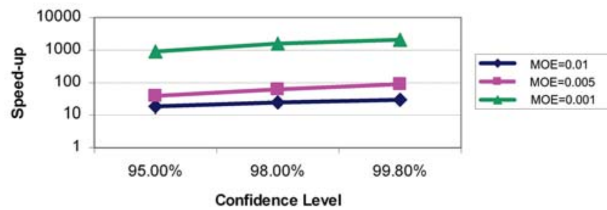

Figure 6: RAVEN speed-up, compared to SFI, for different confidence levels and MOE values.

SFI samples. This way, we can have a more precise comparison between RAVEN and SFI. Even for the smallest sample that we have used, RAVEN works more than 17x faster than SFI, on average. We have also extrapolated the run times for larger samples (Fig. 2) and smaller MOE values, and compared them to RAVEN run times. Speed-up (ratio of SFI to RAVEN run time) is calculated for each workload and we have averaged the speed-up values using geometric mean formula. These averaged speed-ups are shown in Fig. 6.

As shown in this section, RAVEN can estimate the outcomes of soft error candidates in a system mostly within the range indicated by SFI. Due to Fig. 4, only SDC rate in crafty application is out of calculated range of SFI and it is still close to the minimum value of the calculated range of SFI. Our results in Fig. 6 show that RAVEN can estimate the outcome rates 1 to 3 orders of magnitude faster than SFI, depending on the sample size. These calculations are based on the assumption that we use only one computation resource. If we distribute RAVEN processes among **15** computing resources (because we have 15 partitions), we would need, on the average, **253** computing resources for SFI in order to take the same time as RAVEN (for confidence level=95% and MOE=0.01).

4 Discussion

The ideal way for measuring the vulnerability of a system is to calculate the outcome rates for any possible soft error that can happen in a system, which we call *complete error injection*. If we want to compare RAVEN with error injection fairly, we need to perform a complete error injection during the time period that we run RAVEN and compare their outcome rates and run times. Since there are 14,184 flip-flops in IVM that are used for error injection, we need more than 33,600,000 injections for this fair comparison, which is obviously not possible

Table 1: run-time comparison for RAVEN vs. complete error injection.

Application	RAVEN run time (hrs)	Complete injection run time(hrs)	speed-up
bzip2	29.16	1100678	37741
crafty	22.48	566319	25186
gap	13.26	601117	45333
gcc	25.88	762815	29474
gzip	26.97	1645344	61006
mcf	26.30	744092	28297
parser	25.24	576437	22835
vpr	26.90	594782	22107
G. Mean	**23.95**	**767161**	**32026**

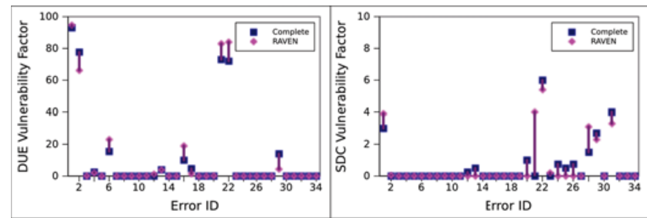

Figure 7: RAVEN vs. complete error injection for 400 cycles and 34 random flip-flops.

to do in a reasonable time. Table 1 shows this comparison between extrapolated complete error injection and RAVEN run time for each workload and the geometric mean of all workloads. This table shows over 32,000x speed-up over the complete error injection.

In this section, we also discuss how RAVEN can be used to design resilient systems. As discussed in previous sections, RAVEN calculates the detection probability of each flip-flop in the system when a workload is running. These detection probabilities can be interpreted as vulnerability factors calculated in error injection, due to their definition. To find the accuracy of the vulnerability factor for each flip-flop in RAVEN, we need to perform complete error injection for those flip-flops and compare their vulnerability factors with the corresponding ones in RAVEN. Since complete error injection for every possible error candidate and every clock cycle is not feasible, we have performed complete error injection for 34 randomly selected flip-flops during a relatively short period (400 cycles) in the *crafty* workload and calculated RAVEN probabilities for the same time period and the same flip-flops, and compared the vulnerability factors calculated using complete error injection and RAVEN. This is shown in Fig. 7.

As can be seen in this figure, RAVEN probabilities have some inaccuracy due to all the sources of inaccuracy that we discussed in previous sections. However, in most of the cases, the results of RAVEN and complete error injection are very

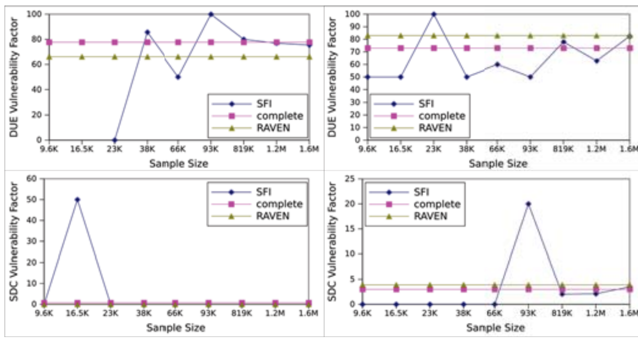

Figure 8: Four examples of vulnerability factor values in complete error injection, SFI, and RAVEN.

close, if not equal. Since we have the complete information during this period for the selected flip-flops, we can simply compute the vulnerability factors obtained by SFI with different sample sizes. We need to generate a random list of error candidates for a desired sample size, and select the errors injected in the flip-flop of interest. We have done this for all the confidence levels and MOEs used in Fig. 2 for 400 cycles and calculated vulnerability factors for each sample size. Interestingly, these vulnerability factors have a large variance over different sample sizes and in some cases, since there is no error injected on the selected flip-flop, we do not have any information about its vulnerability factor. Figure 8 shows this variance for some cases. As an example, for a flip-flop with vulnerability factor equal to 78%, sampling calculates vulnerability factors equal to 0%, 85%, 50%, 100%, 80%, 77%, and 75% for sample sizes equal to 23K, 38K, 66K, 93K, 819K, 1.2M, and 1.6M, respectively. For 9.6K and 16.5K samples, there is no vulnerability factor information since that flip-flop was never chosen. The unstable results in SFI is due to a large MOE in vulnerability factor calculation and insufficient sample size per each flip-flop. According to Eq. 5, if we want to calculate the vulnerability factors in 400 cycles with MOE=0.05 (i.e. range=0.1), we need 196 samples related to each flip-flop. This results in more than 2.7M injections for the whole IVM.

Due to large MOE of SFI for doable sample sizes, deciding on a resilience technique based on SFI vulnerability factors can result in an over- or under-designed resilient system compared to using vulnerability factors from RAVEN.

5 Conclusions

This paper discussed RAVEN, a new technique for estimating the effects of soft errors for a specified workload. RAVEN achieves significant speedup compared to Statistical Fault Injection, with similar estimated rates of the outcomes and more precise estimated vulnerability factors for each flip-flop. Future directions include ways of estimating the vulnerabilities of ultra large-scale systems by utilizing the four orders of magnitude potential speedups with RAVEN compared to complete error injection.

6 Acknowledgments

This work is sponsored in part by a grant from Cisco through Silicon Valley Community Foundation and by Defense Advanced Research Projects Agency, Microsystems Technology Office (MTO), under contract no. HR0011-13-C-0022. The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government. The authors acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing HPC resources that have contributed to the error injection results reported in this paper.

References

[1] Ghazanfar Asadi and Mehdi B Tahoori. An accurate SER estimation method based on propagation probability [soft error rate]. In *Proceedings of Design, Automation and Test in Europe, 2005*, pages 306–307. IEEE, 2005.

[2] C Bottoni, M Glorieux, JM Daveau, G Gasiot, F Abouzeid, S Clerc, L Naviner, and P Roche. Heavy ions test result on a 65nm Sparc-V8 radiation-hard microprocessor. In *IEEE International Reliability Physics Symposium, 2014*. IEEE, 2014.

[3] Hungse Cha, Elizabeth M Rudnick, Gwan S Choi, Janak H Patel, and Ravishankar K Iyer. A fast and accurate gate-level transient fault simulation environment. In *The Twenty-Third International Symposium on Fault-Tolerant Computing, 1993. FTCS-23. Digest of Papers.*, pages 310–319. IEEE, 1993.

[4] Hyungmin Cho, Shahrzad Mirkhani, Chen-Yong Cher, Jacob A Abraham, and Subhasish Mitra. Quantitative evaluation of soft error injection techniques for robust system design. In *50th ACM/EDAC/IEEE Design Automation Conference (DAC), 2013*, pages 1–10. IEEE, 2013.

[5] Abhijit Dharchoudhury, Sung-Mo Kang, Hungse Cha, and Janak H Patel. Fast timing simulation of transient faults in digital circuits. In *Proceedings of the 1994 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 719–722. IEEE Computer Society Press, 1994.

[6] Mahdi Fazeli, Seyed Ghassem Miremadi, Hossein Asadi, and Mehdi Baradaran Tahoori. A fast analytical approach to multi-cycle soft error rate estimation of sequential circuits. In *13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools (DSD), 2010*, pages 797–800. IEEE, 2010.

[7] Daniel Holcomb, Wenchao Li, and Sanjit A Seshia. Design as you see FIT: System-level soft error analysis of sequential circuits. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 785–790. European Design and Automation Association, 2009.

[8] Régis Leveugle, A Calvez, Paolo Maistri, and Pierre Vanhauwaert. Statistical Fault Injection: quantified error and confidence. In *Design, Automation & Test in Europe Conference & Exhibition, 2009. DATE'09.*, pages 502–506. IEEE, 2009.

[9] Xiaodong Li, Sarita V Adve, Pradip Bose, and Jude A Rivers. SoftArch: an architecture-level tool for modeling and analyzing soft errors. In *Proceedings of International Conference on Dependable Systems and Networks, 2005*, pages 496–505. IEEE, 2005.

[10] Shahrzad Mirkhani, Jacob A Abraham, Toai Vo, Hongshin Jun, and Bill Eklow. FALCON: Rapid statistical fault coverage estimation for complex designs. In *IEEE International Test Conference (ITC), 2012*, pages 1–10. IEEE, 2012.

[11] Shubhendu S Mukherjee, Joel Emer, and Steven K Reinhardt. The soft error problem: An architectural perspective. In *11th International Symposium on High-Performance Computer Architecture, 2005. HPCA-11.*, pages 243–247. IEEE, 2005.

[12] Arun A Nair, Lizy Kurian John, and Lieven Eeckhout. AVF stressmark: Towards an automated methodology for bounding the worst-case vulnerability to soft errors. In *43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2010*, pages 125–136. IEEE, 2010.

[13] Andrea Pellegrini, Kypros Constantinides, Dan Zhang, Shobana Sudhakar, Valeria Bertacco, and Todd Austin. CrashTest: A fast high-fidelity FPGA-based resiliency analysis framework. In *IEEE International Conference on Computer Design, 2008. ICCD 2008.*, pages 363–370. IEEE, 2008.

[14] Heather M Quinn, Dolores A Black, William H Robinson, and Stephen P Buchner. Fault simulation and emulation tools to augment radiation-hardness assurance testing. *IEEE Transactions on Nuclear Science*, 60(3):2119–2142, 2013.

[15] Krishnan Ramakrishnan, R Rajaramant, Narayanan Vijaykrishnan, Yuan Xie, Mary Jane Irwin, and Kenan Unlu. Hierarchical soft error estimation tool (HSEET). In *9th International Symposium on Quality Electronic Design, 2008. ISQED 2008.*, pages 680–683. IEEE, 2008.

[16] Pia N Sanda, Jeffrey W Kellington, Prabhakar Kudva, Ronald Kalla, Ryan B McBeth, Jerry Ackaret, Ryan Lockwood, John Schumann, and Christopher R Jones. Soft-error resilience of the IBM POWER6 processor. *IBM Journal of Research and Development*, 52(3):275–284, 2008.

[17] Nicholas J Wang, Justin Quek, Todd M Rafacz, and Sanjay J Patel. Characterizing the effects of transient faults on a high-performance processor pipeline. In *International Conference on Dependable Systems and Networks (DSN), 2004*, pages 61–70. IEEE, 2004.

[18] Xin Xu and Man-Lap Li. Understanding soft error propagation using efficient vulnerability-driven fault injection. In *42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2012*, pages 1–12. IEEE, 2012.