FUSED: A Low-cost Online Soft-Error Detector

Vishal Chandra Sharma, Zvonimir Rakamarić, Ganesh Gopalakrishnan
School of Computing, University of Utah, USA
Email: {vcsharma,zvonimir,ganesh}@cs.utah.edu

***Abstract*—The growth in soft error rates caused by shrinking device geometries and transistor variability can undermine system reliability, requiring cross-layer resilience solutions. In this paper, we make following contributions to this area. First, we introduce a new framework called FUSED in which soft-error detectors are automatically compiled from and inserted into application code through the Rose compilation framework that is widely used in HPC. Our error detectors are based on control-flow tracking through predicate transitions. Second, we develop a new heuristic based on the idea of invalid predicate transitions to identify sensitive-code-blocks causing silent data corruption (SDC) in a program's execution output. New results report in this paper include showing the feasibility of using likely program invariants (in the form of predicate transitions) in realistic code, automation of error detector insertion, use of our empirical findings to diagnose SDC causing code blocks, and evaluation of these techniques on a non-trivial scientific benchmark. Preliminary evaluation on the SuperLU scientific library – a direct solver for sparse and unsymmetric system of linear equations – indicates that our detectors achieve an error detection rate of upto 90.5% while causing an average overhead of only 15.7% in the application runtime.**

I. Introduction

Hardware reliability is increasingly becoming an area of concern, given the variability in transistors and shrinking die size, coupled with the growing scale of systems [1]–[3]. Studies [4] indicate that efficient software level error detectors are important, given the need to minimize the use of the 'always on' hardware-level checkers that can exacerbate energy consumption. Many schemes for synthesizing software-level detectors have been proposed, including the use of *likely program invariants* [5]. This paper builds on our initial proposal in this area—namely the use of *predicate transitions* [6] as likely invariants. We offer the following contributions in this paper:

1) We present a new error-detection framework called FUSED that employs the ROSE compiler infrastructure [7], [8] to instrument the programs for extracting likely program invariants in the form of *predicate transitions* with respect to user-provided training data.
2) We evaluate the efficacy of the *predicate transitions* as the likely invariants in a more realistic setting, namely that of a sequential version of the SuperLU library [9], [10]. We also report a preliminary result on the error-detection rate, and the associated overhead of the FUSED framework.
3) We present a heuristic to identify highly sensitive-code-blocks in the LU factorization routine of the SuperLU library, with a possible future application in optimizing the placement of the detectors.

II. Background & Related Work

Hardware faults evading hardware and micro-architectural level protections cause random bit-flips in the application computational states. These hardware faults introduced can either be permanent or transient in nature. Transient hardware faults (a.k.a. soft-errors) typically persist for a short duration, and are often attributed to cosmic radiations and alpha particles [2], [3]. Transient hardware faults occurring in the CPU state elements can cause application hang, crash, or silent data corruption (SDC) in the execution output, and are harder to detect as compared to permanent hardware faults. This has lead to recent upsurge in the focus towards developing low-cost application-level detectors as an alternative to hardware-level solutions.

There has been an effort to develop application-level detectors by monitoring software-level program properties and extracting likely invariants [5]. Another work involves similar approach for building error detectors in the form of program assertions by analyzing program properties and related rules on a set of dynamic execution traces [11]. Techniques are also proposed to build application-specific detectors by analyzing program properties where majority of the SDC causing soft-errors are visible [12]. Another approach builds architecture-level detectors by tracking history of the processor states with respect to static instructions of a program. The history information is then used for detecting soft-errors occurring in the processor elements [13]. Software-level detectors based on pure control-flow tracking have also been developed [14]. Algorithm based error detection techniques like checksum based approach in matrix-multiplication [15], or techniques for localizing errors in the execution output of a linear solver have also been introduced [16].

Our approach is based on the idea of predicate abstraction used by Ball to derive novel program coverage metrics [17]. We use this approach in building application-level soft-error detectors by extracting likely program invariants in the form of *predicate transitions*.

III. An Overview of Our Approach

Table I shows the tracking of spurious *predicate transitions* to detect soft-errors which cause a control-flow violation (case 2), and a data error (case 3). In the example, labels L0–L5 are the program locations, and PP0–PP4 are the program points. We choose the program conditionals $(x < 5)$ and $(y < 4)$ as the predicates. A predicate state with respect to a program point PP is defined as PP:$\langle \phi_1 \phi_2 \rangle$, where $\langle \phi_1 \phi_2 \rangle$ is a bit-vector. In the bit-vector $\langle \phi_1 \phi_2 \rangle$, ϕ_1 and ϕ_2 represent the

Case 1: Fault-free execution	**Case 2: Faulty execution**	**Case 3: Faulty execution**

L0: int x = 5;
L1: int y = 3;
PP0:
L2: if (x< 5 && y < 4)
PP1:
L3: y = y + 2 ;
PP2:
L4: else
PP3:
L5: x − − ;
PP4:
	L0: int x = 5;
	// bit flip at bit 0 in var x
	// new value of x = 4
	L1: int y = 3;
	PP0:
	L2: if (x< 5 && y < 4)
	PP1:
	L3: y = y + 2 ;
	PP2:
	L4: else
	PP3:
	L5: x − − ;
	PP4:
		L0: int x = 5;
		L1: int y = 3;
		// bit flip at bit 2 in var y
		// new value of y = 7
		PP0:
		L2: if (x< 5 && y < 4)
		PP1:
		L3: y = y + 2 ;
		PP2:
		L4: else
		PP3:
		L5: x − − ;
		PP4:

Program Output: x=4 , y=3	Program Output: x=4 , **y=5** (SDC)	Program Output: x=4 , **y=7** (SDC)
Predicate Transitions: PP0:FT → PP3:FT PP3:FT → PP4:TT	Predicate Transitions: **PP0:TT → PP1:TT** **PP1:TT → PP2:TF**	Predicate Transitions: **PP0:FF → PP3:FF** **PP3:FF → PP4:TF**

TABLE I: A Motivating Example

boolean values of the chosen predicates $(x < 5)$ and $(y < 4)$ respectively at the program point PP. Using this definition, the concrete predicate state observed at the program point PP0 is PP0:FT as the predicates $(x < 5)$ and $(y < 4)$ evaluate to *false* and *true* respectively.

A predicate transition comprises of a current state, a next state, and a transition from the current to the next state. For example, in case 1, the concrete predicate transition with respect to PP0 and PP3 is represented as PP0:FT → PP3:FT, where PP0:FT and PP3:FT are the concrete predicate states at the program points PP0 and PP3 respectively. In case 2 and 3, a bit-flip (induced by a soft-error) causes SDC in the program output, and spurious predicate transitions (both highlighted in red color) which are absent in the fault-free execution instance shown in case 1. In section IV, we describe our error detection framework in detail, which uses this idea of tracking erroneous predicate transitions to detect soft-errors at the application runtime.

IV. FUSED: A Soft-Error Detector

A. Design

The FUSED framework performs source-level instrumentation for extracting the likely invariants, and inserting the detectors into a target program. All instrumentations are done by modifying the abstract syntax tree (AST) representation of the program. Figure 1 shows the two modes of operation for the FUSED framework: (i) the *profiler* mode, and (ii) the *detector* mode. The *profiler* mode is used to extract the likely program invariants in the form a list of concrete predicate transitions. The *detector* mode is used for building and inserting soft-error detectors into the target program.

B. Generating Likely Invariants

In the *profiler* mode, FUSED generates a separate *profiler* function for each function in the target program. The *profiler* function takes program point locations, and the concrete values of the boolean program conditionals of the target function as its inputs arguments. The generated *profiler* function definition is inserted into the target program. Function invocation code to call the *profiler* function is also inserted at user-selected location in the target function. Fault-free executions of the instrumented program for a training input set, produces a set of concrete predicate transitions. We call this set of concrete predicate transitions as the *valid predicate transitions* which are used as the likely invariants for run-time soft-error detection.

C. Building Detectors

In the *detector* mode, FUSED generates a new *detector* function for each function in the target program. The *detector* function accepts the same set of input arguments as the *profiler* function, and is inserted into the target program at user-defined program locations. Detectors in the form of function invocation code for calling the *detector* function are also inserted into the target function. These detectors when invoked at runtime, evaluates the currently observed concrete predicate transition, and checks whether it is present in the list of *valid predicate transitions*. A concrete predicate transition missing from the list of *valid predicate transitions* signals the detection of a soft-error, and is called an *invalid predicate transition*. Section IV-D presents a heuristic to identify sensitive-code-blocks using the *invalid predicate transitions*. In our experimental setup, we use KULFI, an instruction level fault injector for simulating soft-errors [6]. Our approach of using likely invariants (in the form of *valid predicate transitions*), suffers from a natural drawback of false alarms. In order to reduce false alarms for our experimentations, we extract the set of *invalid predicate transitions* observed during fault-free executions and add them to the set of *valid predicate transitions*. In real world scenario, one good strategy for reducing false alarms is to use

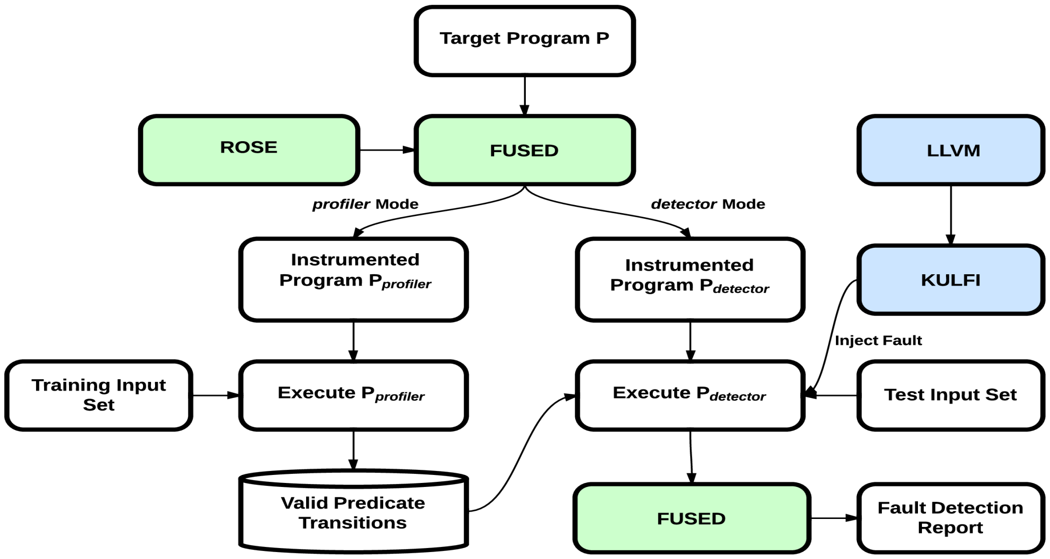

Fig. 1: Workflow of FUSED Framework

a larger sample size for the training input set to bring down the false-alarm rate to an acceptable limit.

D. Identifying Sensitive Code Blocks

Algorithm 1 presents a heuristic to identify sensitive-code-blocks which under influence of soft-errors are most likely to cause SDC in the program output. A set of *top invalid predicate transitions* δ^T is selected from the set of *invalid predicate transitions* δ_I based on their frequency of occurrence during the faulty program executions, by invoking the FUSED interface *GetTopInvalidTransitions()*. The elements in the set δ^T is further sorted in the order of their places of occurrences in the program $P_{detector}$ to obtain an ordered set δ^T_{sorted}. For each predicate transition δ_i in the set δ^T_{sorted} which has an enclosing code block with one or more program statements, the corresponding program statements are added to the set $Stmt_i$. Each program statement in $Stmt_i$ is syntactically analyzed to see if they can influence the runtime concrete boolean values of the predicates in the set Φ. The program statements affecting the concrete boolean values of the predicates, are added to the set $Stmt$. The set $Stmt$ is considered as the set of program statements in the program P which are most vulnerable to soft-errors.

Algorithm 1 – IdentifyVulnerableCodeBlocks

```
$\delta^T \leftarrow GetTopInvalidTransitions(\delta_I)$
$\delta^T_{sorted} \leftarrow SortByProgramLocation(\delta^T)$
for all $\delta_i$ in $\delta^T_{sorted}$ do
    $Stmt_i \leftarrow GetEnclosingCodeBlock(\delta_i)$
    for all $stmt_j$ in $Stmt_i$ do
        $Var \leftarrow GetCommonVar(stmt_j, \Phi)$
        if $Var$ is empty then
            $Stmt \leftarrow Stmt \cup stmt_j$
        end if
    end for
end for
return $Stmt$
```

V. An Empirical Case Study

A. Strategy

Table II shows the details of the set of matrices used as the input sets for the experimentations, and its classification based the problem domains. In our experimentations, we carry out 20 fault injection campaigns for each of the input category. Each of these fault injection campaigns comprises of 1000 runs which sums upto 20,000 fault injection experiments for each of the 5 input categories, to obtain a statistically significant result.

Before starting the fault injection campaigns, the outputs from the fault-free executions of the SuperLU library with all the test inputs are recorded as *golden* program outputs. We also profile the set of *valid predicate transitions* by running the instrumented version of the SuperLU library with profilers inserted after its every program statement of *dgstrf()* function. The training input matrix for the *profiler* mode of execution is chosen at random from the respective input category.

During each run of a fault injection campaign, an instrumented version of the SuperLU library with detectors inserted after every program statement in its *dgstrf()* function, is executed. The test input matrix for the *detector* mode of execution is chosen at random from the respective input category and is different from the one used during the *profiler* mode of execution. During the program execution, the detectors use the previously generated set of *valid predicate transitions* to detect soft-errors injected using KULFI at the application runtime.

The program output recorded from each run is compared with its corresponding *golden* output, and then classified into one of the following categories:
Benign: When the program outputs from the fault-free and the

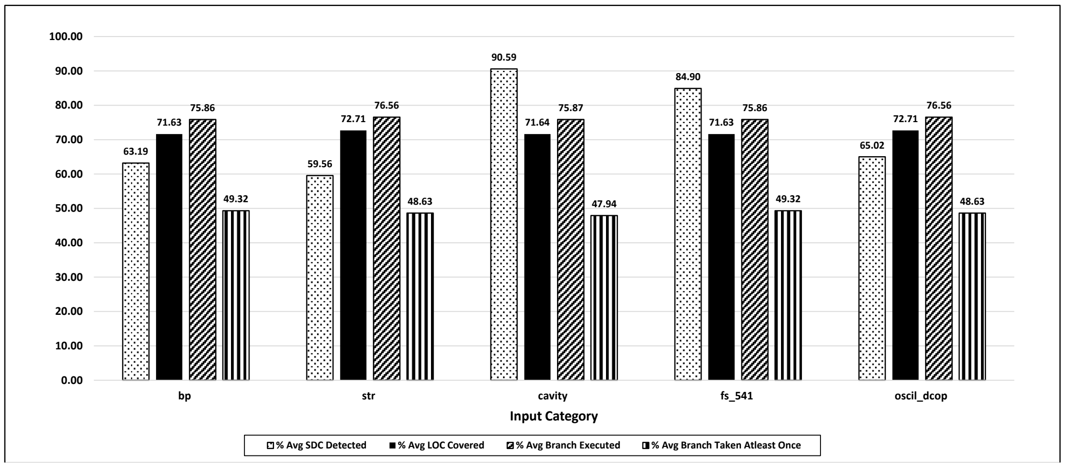

Fig. 2: Soft-Error Detection Rate & Coverage Metrics

faulty executions are identical.

Segmentation Fault: When an out-of-bound access during program execution causes program crash or hang resulting in incomplete or no program output.

Silent Data Corruption (SDC): When the program output of the faulty execution differs from its corresponding *golden* output, and no crash is observed during the program execution. Finally, FUSED compiles the fault detection statistics at the end of all those runs where SDC is observed in the program output.

Problem Domain	Input Category	AvgDIC	Minimum Input Size
Optimization I	bp	262k	822 x 822
Optimization II	str	122k	363 x 363
Computational Fluid Dynamics	cavity	92k	317 x 317
2D/3D	fs_541	152k	541 x 541
Circuit Simulation	oscil_dcop	135k	430 x 430

TABLE II: Input Classification

B. Fault Model

During each run of the SuperLU library, we inject exactly one single-bit fault into a data register of a randomly chosen LLVM-level dynamic instruction using the instruction-level fault injector KULFI. The dynamic LLVM-level instruction is chosen with a probability of $\frac{1}{N}$, where N is the total number of LLVM-level dynamic instructions enumerated. This fault model has been the model of choice for various resilience studies done in the recent past [6], [18].

C. Result

We evaluate the FUSED framework by applying it on sequential SuperLU scientific library. We choose the training and the test input sets for our experimentations from a variety of problem domains available in the University of Florida sparse matrix collection [19]. For our experimentations, we restrict the placement of the profilers and the detectors to the *dgstrf()* function in the SuperLU routine which is the core function responsible for performing LU factorization. In our experimentations, we consider placing the profilers and the detectors after every program statements of the *dgstrf()* function. Choosing this option helps us with our preliminary study of analyzing the influence of different program properties on the set of concrete predicate transitions observed in the faulty and fault-free execution traces. For future studies, we plan to learn from our current study to develop heuristics for optimizing the placement of the detectors.

In Table II, AvgDIC is the average dynamic instruction count. In general, the SuperLU library is resilient with a very low average rate of SDC in the execution output, ranging between 7.2 and 11.7 per 1000 fault injection experiments. Results in figure 2 are calculated by averaging the individual results obtained from the 20 fault injection campaigns per input category, and are statistically significant as they follow the three-sigma rule. The highest average SDC detection rate of **90.59%** is observed for *cavity* input category. The average SDC detection rate across all the 5 input categories, stands at **72.6%** with an average execution overhead of **15.7%**. An average false alarm rate of 35.6% is observed during the experimentations which is addressed using the method described in section IV-C. Table III shows the list of sensitive-code-blocks obtained by applying the heuristic presented in Algorithm 1. Sensitive-code-blocks are the code regions in the program which in the presence of soft-errors, are most likely to cause SDC in the program output. Interestingly, the list of most sensitive program statements observed across all input categories is identical.

Line#	Program Statement
280	kcol = relax_end[jcol];
306	k = xa_begin[icol];
330	k = jcol + 1;
349	k = (jj - jcol) * m;
384	jcol += panel_size;

TABLE III: Sensitive Code Blocks

Clearly, these are very preliminary results meant for the initial evaluation of the FUSED framework. In future, we plan to validate our approach on a much larger scale.

VI. Conclusions and Future Work

We presented a new soft-error detection framework FUSED which automatically synthesizes and inserts program-level detectors. We demonstrated its effectiveness by evaluating it on a real-world example of SuperLU library. We also presented a preliminary version of a heuristic for identifying sensitive-code-blocks at source-code level. There are various avenues to explore in near future. We plan to use the proposed heuristic to identify sensitive-code-blocks for optimizing placement of the detectors thereby also reducing the execution overhead incurred during the run-time error detection. Also, developing techniques to refine selection of more meaningful predicates would be a definite area to pursue. Using static analysis techniques to extract sound invariants for error-detection will also be explored. Finally, we plan to apply this framework on further broader set of real-world examples, and on much larger sample size of test and training input sets.

VII. Acknowledgement

We would like to thank Greg Bronevetsky and anonymous reviewers of this paper for their insights and feedback.

References

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

[2] J. Rivers and P. Kudva, "Reliability challenges and system performance at the architecture level," *in IEEE Design Test of Computers (DTC)*, vol. 26, no. 6, pp. 62–73, 2009.

[3] S. Michalak, K. Harris, N. Hengartner, B. Takala, and S. Wender, "Predicting the number of fatal soft errors in los alamos national laboratory's asc q supercomputer," *in IEEE Transactions on Device and Materials Reliability (DMR)*, vol. 5, no. 3, pp. 329–335, 2005.

[4] J. A. Rivers, M. S. Gupta, J. Shin, P. N. Kudva, and P. Bose, "Error Tolerance in Server Class Processors," *in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (CADICS)*, vol. 30, no. 7, pp. 945–959, 2011.

[5] Sahoo, Swarup Kumar and Li, Man-lap and Ramachandran, Pradeep and Adve, Sarita V. and Adve, Vikram S. and Zhou, Yuanyuan, "Using likely program invariants to detect hardware errors," in *IEEE International Conference on Dependable Systems and Networks (DSN)*, 2008, pp. 70–79.

[6] V. C. Sharma, A. Haran, Z. Rakamarić, and G. Gopalakrishnan, "Towards formal approaches to system resilience," in *IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)*, 2013, to appear.

[7] D. J. Quinlan, "Rose: Compiler support for object-oriented frameworks," *in Parallel Processing Letters (PPL)*, vol. 10, no. 2/3, pp. 215–226, 2000.

[8] "ROSE: A Compiler Infrastructure," http://rosecompiler.org/.

[9] X. Li, J. Demmel, J. Gilbert, iL. Grigori, M. Shao, and I. Yamazaki, "SuperLU Users' Guide," http://crd.lbl.gov/~xiaoye/SuperLU/, Lawrence Berkeley National Laboratory, Tech. Rep. LBNL-44289, 1999.

[10] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu, "A supernodal approach to sparse partial pivoting," *in SIAM Journal on Matrix Analysis and Applications (SIMAX)*, vol. 20, no. 3, pp. 720–755, 1999.

[11] K. Pattabiraman, G. Saggese, D. Chen, Z. Kalbarczyk, and R. Iyer, "Automated derivation of application-specific error detectors using dynamic analysis," *in IEEE Transactions on Dependable and Secure Computing (DSC)*, vol. 8, no. 5, pp. 640–655, 2011.

[12] S. K. S. Hari, S. V. Adve, and H. Naeimi, "Low-cost program-level detectors for reducing silent data corruptions," in *IEEE International Conference on Dependable Systems and Networks (DSN)*, 2012, pp. 1–12.

[13] P. Racunas, K. Constantinides, S. Manne, and S. Mukherjee, "Perturbation-based fault screening," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2007, pp. 169–180.

[14] N. Oh, P. P. Shirvani, E. J. Mccluskey, and L. Fellow, "Control-flow checking by software signatures," *in IEEE Transactions on Reliability*, vol. 51, no. 2, pp. 111–122, 2002.

[15] T. Davies and Z. Chen, "Correcting soft errors online in lu factorization," in *International Symposium on High-performance Parallel and Distributed Computing (HPDC)*, 2013, pp. 167–178.

[16] Sloan, Joseph and Kumar, Rakesh and Bronevetsky, Greg, "An algorithmic approach to error localization and partial recomputation for low-overhead fault tolerance," in *IEEE International Conference on Dependable Systems and Networks (DSN)*, 2013, pp. 1–12.

[17] T. Ball, "A Theory of Predicate-Complete Test Coverage and Generation," in *International Conference on Formal Methods for Components and Objects (FMCO)*, 2005, pp. 1–22.

[18] M. de Kruijf, S. Nomura, and K. Sankaralingam, "Relax: An architectural framework for software recovery of hardware faults," in *International Symposium on Computer Architecture (ISCA)*, 2010, pp. 497–508.

[19] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *in ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, pp. 1–25, 2011.