Identification of Critical Variables using an FPGA-based Fault Injection Framework

Andreas Riefert Jörg Müller Matthias Sauer Wolfram Burgard Bernd Becker

Albert-Ludwigs-University Freiburg
Georges-Köhler-Allee 051
79110 Freiburg, Germany
{ riefert | muellerj | sauerm | burgard | becker }@informatik.uni-freiburg.de

***Abstract*— The shrinking nanometer technologies of modern microprocessors and the aggressive supply voltage down-scaling drastically increase the risk of soft errors. In order to tackle this problem, we propose an FPGA-based fault injection framework which is able to identify the most critical parts of a system running in its native environment. Experimental results demonstrate, that our approach significantly reduces the number of critical calculation errors in the evaluated applications.**

I. Introduction

In recent research, single-event upsets (SEU) have been shown to occur in modern memory elements with a rate of about 5000 FIT per Mbit [1].This induces upset events at an interval of days or even hours in modern microprocessors.

On the one hand, there exist several hardware and software protection schemes to deal with SEUs and single-event transients (SETs). However, applying these techniques to all memory elements of a circuit or to an entire software application, respectively, results in substantially higher costs or drastically reduced performance. On the other hand, modern microprocessors implicitly mask a lot of the transient faults, i.e., many faults have no effect [2]. Additionally, certain software applications already partially imply fault tolerance.

In this paper we present a fault injection framework, which is able to identify the most critical registers of a given microprocessor and the most critical variables of an arbitrary application. Thereby, we implemented an efficient and lightweight protection scheme by combining the implicit fault tolerance of applications with explicit fault detection and correction techniques.

We demonstrate the applicability of our approach in experiments with an efficient error correction mechanism applied to the identified critical variables of a state estimation application.

The rest of the paper is organized as follows: Sections II and III describe our fault injection framework and the software applications under test. Section IV presents our first experimental results and Section V concludes the paper.

II. System Description

Our fault injection framework is an extension of [3]. The novel contribution comprises the identification of the critical software variables. The flow described in this section works as follows: First fault injection experiments are executed, then the faulty runs are used to identify the critical variables, which are finally protected by a software approach.

The framework consists of a Generic Experiment Manager and an FPGA-based fault injector. The Generic Experiment Manager is executed on a desktop PC and is responsible for running the experiments. On the FPGA-side, faults are injected into a target processor by the fault injection module.

The experiment manager generates a list of faults to be injected and passes them to the fault injector on the FPGA. It transmits the selected application and the input data and starts the execution on the target processor. As soon as the target processor returns the application results, they are evaluated by the Generic Experiment Manager. If the target processor does not respond within a certain time, we regard this as a Total System Failure (TSF) and terminate the run.

The fault injection into the registers of the target processor is implemented by duplicating all registers and connecting these new *Shadow Registers* to a scan chain. This enables us to shift a fault to an arbitrary register.

Experimental evaluation shows that faults in the control structure and memory management unit of the target processor cause a lot of TSFs and calculation errors. Possible solutions are the extension of registers with ECC [4] or the use of Dual Interlocked CEll flip flops (DICE) [5]. These protection schemes induce higher chip costs, thus it is not practical to protect all registers. As faults in the register file lead to only a few TSFs, we propose to protect the contained registers by software redundancy.

As a protection of the whole application code would result in a substantial performance overhead, we identify the most critical parts of the application under test, i.e., the most critical variables with respect to the resulting calculation error. Therefor we record the call stack of the program at the time of the fault injection. This allows us to determine all source code statements and variables, which were probably affected by the fault. By analyzing the faulty experiment runs, we created a ranking of critical variables.

To experimentally show the significance of the identified critical variables, we use a basic error correction and error detection scheme for these variables, similar to [6]. All corresponding computations are duplicated. A check mechanism, implemented with seven assembler instructions, compares the results and, if necessary, repeats the computations.

III. Bayes Filter State Estimation

In the experiments presented in this paper, we consider the application of estimating the three-dimensional orientation $\mathbf{x}$ of an inertial measurement unit (IMU). This software application is designed for operation on noisy sensor data and therefore partially implies tolerance to transient faults. We apply the Bayesian filtering scheme [7]. In particular, we recursively estimate the posterior probability density $p(\mathbf{x}_t \mid \mathbf{z}_{1:t}, \mathbf{u}_{1:t})$ of the state $\mathbf{x}_t$ at time t given all the noisy data, i.e., the corrective measurement data $\mathbf{z}_{1:t}$ and the predictive data $\mathbf{u}_{1:t}$ up to time t. We evaluate two implementations of the Bayesian filtering scheme: the extended Kalman filter (EKF) [8] and the particle filter [9].

IV. Experimental Results

A. Experimental Setup

In our experiments we use the *ourMIPS* processor [10], [11], which runs on Altera Cyclone II FPGA starter boards [12]. However, our approach could be also applied to other microprocessors. We examined two filters for sensor data fusion namely the EKF and the particle filter as described in Section III. The input data was collected with an IMU attached to a miniature blimp robot which was observed by an accurate optical motion capture system for position and orientation ground truth information. Under fault injection, we consider the root mean square error (RMSE) of a particular run as a critical error if it is more than 0.5° higher than the fault free RMSE.

To identify the critical variables of an application we first execute 5000 runs, where we inject one fault per run. This gives us the required data for our analysis as described in Section II. In a second step, we implement the checker mechanism (Section II) for the identified critical variables. We run several experiments, where we inject faults into all registers of the register file except five registers, which are associated with the control structure of the processor. This serves as a validation of the previous step.

B. Critical Variables

Figure 1 shows the ranking of critical variables for the EKF, which was created considering only the critical errors. This approach is reasonable, as the considered applications calculate a state estimate from noisy measurement data, which usually can be used, even if it slightly deviates from the fault free computation result.

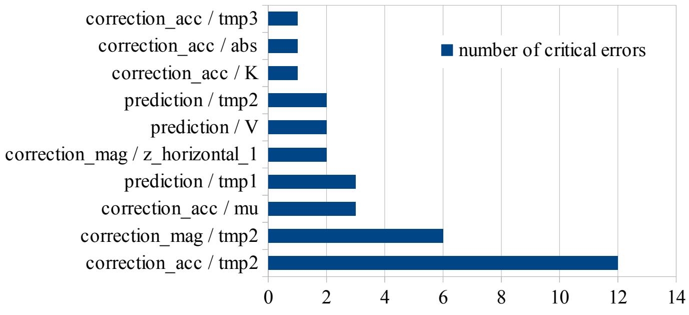

Figure 1. The critical variable statistics created counting only the critical errors of the EKF in 5000 runs. Each bar is labeled with the function the variable appears in and the variable itself.

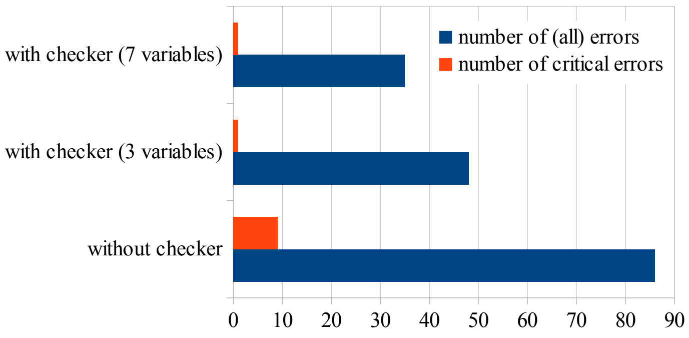

Figure 2. Comparison of different protection levels of the EKF application in 7000 runs.

The evaluation of the particle filter indicates a substantial fault tolerance as the algorithm did not produce any critical calculation errors within 7000 runs. This is reasonable as the particle filter comprises only a few critical control parts and therefore is robust to the injected faults.

C. Experimental Evaluation

We evaluated our approach by protecting the 3 most critical and the 7 most critical variables according to the statistics of the critical errors. The first version increases the runtime by 18%, the second one by 27%. Figure 2 shows the results of the fault injection experiments of the unprotected version compared to that of our two protected versions of the EKF. The number of all calculation errors continuously decreases as the protection level increases. The number of critical errors is decreased from nine in the unprotected case down to one in both protected versions. This is reasonable as the critical variables with the ranks 4 to 7 still produce a certain amount of calculation errors with respect to all deviations, but only a small number of critical errors (see Figure 1). According to our experimental results, the protection of the 3 most critical variables is sufficient in order to reduce the critical calculation errors to a tolerable amount. As the underlying distribution for the number of errors is a binomial distribution, we can test for the improvement using Fisher's exact test. At a 5% confidence level, both versions with software checker have a significantly lower error rate than the unprotected version.

Therefore, we have significantly increased the reliability of our application by protecting only a small part of the source code.

V. Conclusions

In this paper, we presented an FPGA-based fault injection framework which enables a user to identify the most critical registers of an entire microprocessor and the most critical variables of the executed software application. Our experiments show, that the criticality of different parts of a system with regard to system crashes and calculation errors can substantially vary. Therefore protecting only the identified critical parts of a system allows to implement a lightweight and efficient error correction scheme. As our approach is general, it could be easily applied to other microprocessors or applications.

Acknowledgements

Parts of this work were supported by the German Research Foundation (DFG) under grant GRK 1103.

References

[1] *Soft Errors in Electronic Memory - A White Paper.* http://www.tezzaron.com/about/papers/soft_errors_1_-1_secure.pdf.

[2] N. Wang, J. Quek, T. Rafacz, and S. Patel, "Characterizing the effects of transient faults on a high-performance processor pipeline," in *International Conference on Dependable Systems and Networks*, pp. 61–70, 2004.

[3] M. Sauer, V. Tomashevich, J. Müller, M. Lewis, A. Spilla, I. Polian, B. Becker, and W. Burgard, "An FPGA-based framework for run-time injection and analysis of soft errors in microprocessors," in *International On-Line Testing Symposium*, pp. 182–185, 2011.

[4] A. Bouajila, J. Zeppenfeld, W. Stechele, and A. Herkersdorf, "An architecture and an FPGA prototype of a reliable processor pipeline towards multiple soft- and timing errors," in *IEEE International Symposium on Design and Diagnostics of Electronic Circuits & Systems*, pp. 225–230, 2011.

[5] A. Maru, H. Shindou, T. Ebihara, A. Makihara, T. Hirao, and S. Kuboyama, "DICE-based flip-flop with SET pulse discriminator on a 90 nm bulk CMOS process," in *IEEE Transactions on Nuclear Science*, pp. 3602–3608, 2010.

[6] M. Rebaudengo, M. S. Reorda, M. Torchiano, and M. Violante, "Soft-error detection through software fault-tolerance techniques," in *International Symposium on Defect and Fault Tolerance in VLSI Systems*, pp. 210–218, 1999.

[7] S. Thrun, W. Burgard, and D. Fox, *Probabilistic Robotics.* MIT Press, 2005.

[8] Y. Bar-Shalom, T. Kirubarajan, and X. Li, *Estimation with Applications to Tracking and Navigation.* John Wiley & Sons, Inc., 2002.

[9] F. Dellaert, D. Fox, W. Burgard, and W. Thrun, "Monte carlo localization for mobile robots," in *Proc. of the IEEE Int. Conf. on Robotics & Automation (ICRA)*, pp. 1322–1328, 1999.

[10] B. Becker and P. Molitor, *Technische Informatik: Eine einführende Darstellung.* Oldenbourg Wissenschaftsverlag, 2008.

[11] *MIPS32 Architecture For Programmers Volume II: The MIPS32 Instruction Set*, version 2.0, june 9, 2003 ed., 2003.

[12] *Cyclone II FPGA Starter Development Kit.* http://www.altera.com/products/devkits/altera/kit-cyc2-2C20N.html.