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 cope with this risk efficiently, selective hardware and software protection schemes are applied. In this paper, we propose an FPGA-based fault injection framework which is able to identify the most critical registers of an entire microprocessor. Furthermore, our framework identifies critical variables in the source code of an arbitrary application running in its native environment. We verify the feasibility and relevance of our approach by implementing a lightweight and efficient error correction mechanism protecting only the most critical parts of the system. Experimental results with state estimation applications demonstrate a significantly reduced number of critical calculation errors caused by faults injected into the processor.**

I. Introduction

The main source of transient faults in modern digital circuits are cosmic rays or charged α-particles. These phenomena cause local electric noise which can lead to an inversion of the logical value on a line or storage element. If such an event takes place in a storing element, the fault is saved and persists in the circuit until the corresponding memory element is overwritten again [1].

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 [2]. As modern microprocessors tend to have more and more memory bits, e.g., larger caches, this induces upset events at an interval of days or even hours. Obviously, such error rates cannot be neglected, especially in safety-critical applications.

On the one hand, there exist several hardware and software protection schemes to deal with SEUs and single-event transients (SETs). In hardware, error correcting codes (ECC) and additional registers with a delayed clock have been shown to reliably protect a microprocessor [3]. On a lower level, DICE-based flip-flops with an SET pulse discriminator are also able to tolerate SEUs and SETs [4]. Software approaches like *SWIFT* detect transient faults by extending the original program with new validating instructions on the assembler level [5]. 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 [6]. Often, plenty of the available registers of a processor are only rarely used by the compiler. Additionally, certain software applications already partially imply fault tolerance. For example, the state estimation techniques applied in many robotic systems fuse noisy measurement data to a robust state estimate [7] and are able to tolerate some faults in the data.

In this paper we present a fault injection framework, which allows to identify the critical parts of a system in its real environment. Thereby, we can implement efficient and lightweight protection schemes by combining the implicit fault tolerance of applications with explicit fault detection and correction techniques. The focus is on the reduction of the number of silent data corruptions, as they are hard to detect.

Our approach is able to automatically identify the most critical registers of a given microprocessor and the most critical variables of an arbitrary application. In particular, our framework tracks a fault from its injection into a register up to the source code, which is currently executed. 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: The related work is discussed in Section II. Sections III and IV describe our fault injection framework and the software applications under test. Section V presents our first experimental results and Section VI concludes the paper.

II. Related Work

A significant amount of research has been carried out on studying the effects of real soft errors [1], [8] and their simulation and emulation. Bouajila et al. [3] investigate the Leon SPARC V8 processor. The fault injection is simulated in ModelSim and emulated on an FPGA with a linear feedback shift register (LFSR). The Very Long Instruction Word processor r-VEX is implemented on an FPGA and faults are injected into the instruction register, the register file and the SRAM of the processor by Sterpone et al. [9]. Sartori et al. [10] emulate soft errors by utilizing

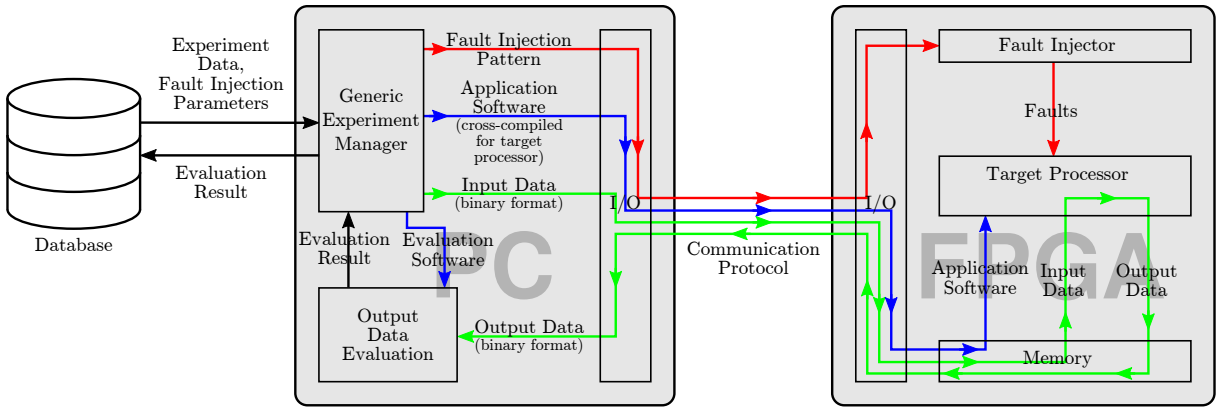

Figure 1. System overview

an FPGA, but faults are only injected into the output of the floating point unit of a Leon 3 processor. In contrast to the techniques mentioned above, our framework allows the user to enable or disable the fault injection for every individual register of the examined microprocessor.

There are several approaches, that identify critical parts of a system using analytical methods instead of fault injection experiments. Seshia et al. [11] proposed to use formal verification tools to determine, whether certain latches in a circuit have to be protected or not. A common method for estimating the criticality of different parts of a system is used by Bergaoui et al. [12]. They focus on determining the criticality of the registers in the register file of a Leon 2 processor. Criteria like lifetime, the number of functional dependencies or conditional branches are used for each register to calculate an overall value, which indicates the criticality of the corresponding register. Nakka et al. [13] utilize this idea in order to identify the critical variables in an application with regard to a failure of the system. Finally, Portela et al. [14] evaluate the efficiency of an algorithm which adds checkpoints in the software application in order to detect faults in the control flow. Additionally, their algorithm tries to protect only the critical parts of the application to reduce the runtime overhead.

Compared to these approaches, our method emulates the effect of transient faults in the application running in its natural environment. Thereby, it tracks each injected fault to the currently executed source code. This allows us to associate faults in specific variables with calculation errors in the application result and to evaluate arbitrary complex algorithms.

III. System Description

Our fault injection framework is an extension of [15]. 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.

Figure 1 gives an overview over the fault injection framework. The Generic Experiment Manager is executed on a desktop PC and is responsible for running the experiments which are scheduled in the database. On the FPGA-side, faults are injected into the target processor by the fault injection module.

The Generic Experiment Manager is responsible for the initialization of the experiments and the evaluation of their results. Thereby, the cross-compiled application as well as the input data and the output data evaluation library are obtained from the database, so that the Generic Experiment Manager can run arbitrary applications.

An experiment consists of multiple runs with the same parameters. A user can specify the examined application and several parameters for the fault injection like the fault rate or the registers where faults should be injected. 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.

All applications and their corresponding evaluation functions are stored in dynamic linked libraries. This facilitates the simple addition of new applications. Furthermore, the experiments and their results are stored in the database, so that it is possible to run the experiments in parallel on several computers and FPGAs.

In our experiments, we use the *ourMIPS* processor [16], which is based on the MIPS instruction set [17]. However, our approach could be also applied to other microprocessors. For convenience, we additionally implemented an ourMIPS simulator, on which the user can execute the applications to ensure their correct functionality.

A. Fault Injection Into Registers

The fault injector on the FPGA stores the received fault injection pattern from the Generic Experiment Manager. Each fault in this list consists of a fault injection time and a register ID. The injection time specifies the point of time in clock cycles to inject the fault whereas the register ID determines a certain bit of a register as the fault location. In order to enable the fault injection into the registers of the ourMIPS we duplicate each register of the processor. These new *Shadow Registers* are connected to a scan chain. This scan chain allows the fault injector to shift a fault, which is represented by a logical '1', to the desired register bit. If the target register is written in the clock cycle of the fault injection, we invert the new value at the specified bit and write this manipulated value into the register. If not, we analogously manipulate the current value of the register and store it. This fault model is reasonable because the wrong logical value induced by a SEU can get stored in a register until it is overwritten with a new value.

With the tools presented so far it is possible to inject faults into certain modules or registers of the ourMIPS. This allows us to experimentally evaluate the criticality of these components with regard to TSFs and calculation errors in the output data.

The ourMIPS consists of four main modules: the control structure, the memory management unit, the ALU and the register file. As previous experiments show, faults in the control structure and in the memory management unit cause a lot of TSFs and calculation errors. This is not surprising since these modules control the processor, the program and the data memory, respectively. Registers contained in these critical modules, like the program counter, require protection in order to increase the overall processor reliability.

Possible solutions are the extension of registers with ECC [3] or the use of Dual Interlocked CEll flip flops (DICE) [4]. These protection schemes induce higher chip costs, thus it is not practical to protect all registers. As our experiments show, faults in the register file result only in a few TSFs. Five of the registers are responsible for most of them as they are also associated with the control structure of the processor, e.g., the stack pointer. The other 27 registers are mainly correlated with the data flow. Therefore we propose to protect them selectively by software redundancy.

B. Identification of Critical Variables

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. For this purpose we trace a fault from its injection back to the source code which is currently executed, the variable which is currently calculated respectively.

First, we have to determine the call stack of the application under test at the time of the fault injection. We need the complete call stack and not only the current program counter value for a simple reason: Assume two variables 'a' and 'b', whose values are both calculated by a function 'f' and a fault, which is injected during the execution of 'f'. If we want to know which variable was affected by the faulty computation of 'f' we need to know whether it was "called" by 'a' or 'b'. Since 'f' itself could call other functions, we have to keep track of the whole call stack. For this purpose we extend the ourMIPS with a lookup table which is supposed to store the call stack for each injected fault.

If we encounter a function call, which is indicated by a *Jump and link* instruction in the MIPS instruction set, we store the corresponding program counter value in the lookup table. In case a function returns without a fault being injected, we delete the corresponding program counter value from the lookup table. If a fault is injected, we also write the current program counter value into the lookup table and add a stop sign. This is necessary to separate the call stacks of different faults. After the application under test is completed and has returned its results, we read all values from the lookup table.

In our fault tracing procedure, we associate each program counter value from the call stack with its corresponding C statement in the source code of the application under test. For this purpose we utilize the tool *objdump*, which is part of the GNU Binutils. This tool is able to create an interleaved representation of the C source code and the corresponding assembler instructions. As the applications under test commonly consist of several source files, we have to create a linker map, which allows us to determine the position of each source file in the executable file. Hereby, we are able to associate each stored program counter value with the appropriate C statement. In the next step we have to check, whether the found C statement assigns a new value to a variable. This is the case when the statement either contains an assignment operator or a function, whose parameters include a variable that is passed per reference and modified inside the function.

The described mechanism is able to trace an injected fault back to a software variable where the possibly faulty value is stored. To identify the most critical variables of our application under test, we execute several runs with one injected fault each. If a run returns a faulty result, we identify all the variables that were modified during all the steps of the call stack. We store these variables in a global list together with a counter. Each time a variable is encountered, its counter is increased. Consequently with a reasonably sized set of faulty runs we are able to create a ranking of variables with respect to their criticality. Here, we define the criticality as the number of faulty runs the variable was associated with.

Figure 2 illustrates the described flow with a small source code example, which was extracted from our evaluated

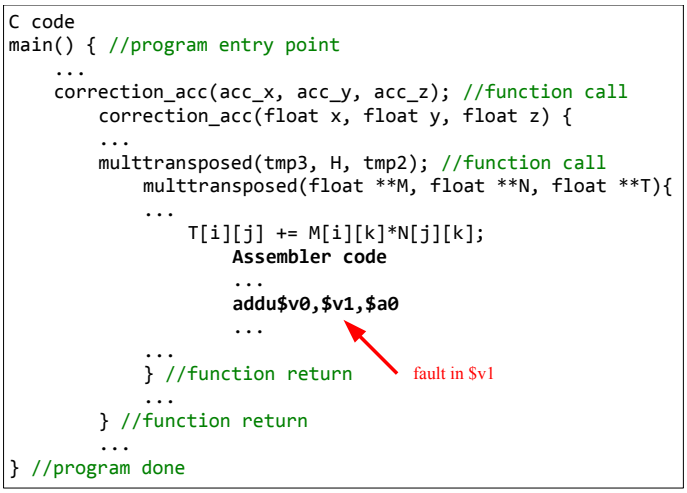

Figure 2. Example for the evaluation of an injected fault

application. The application starts its execution in the `main` function, which makes a function call to `correction_acc`. This function calls `multtransposed`, which contains a C statement that calculates the values of the matrix `T`. This statement, like all other C statements in the source code, is realized by a set of assembler instructions. One of these instructions is shown in Figure 2. It simply adds the values of registers `$v1` and `$a0` and stores the result in `$v0`. If we assume, that a fault is injected in `$v1` during the execution of this instruction, consequently a faulty value is stored in `$v0`. This implicates a faulty value for the corresponding entry of `T`, which is why we add `T` to our global list of critical variables. Now we go one step up in the call stack to the function `multtransposed`. The value of `tmp2` is modified by this function and therefore this variable is also added to the global list. As the functions `correction_acc` and `main` do not modify any values of their arguments, our algorithm returns a list with the variables `tmp2` and `T`, each with a counter value of one.

C. Fault Correction by Software Redundancy

To experimentally show the significance of the identified critical variables, we use a basic error correction scheme for these variables, similar to [18]. Firstly, we duplicate each considered variable and its computations in the C source code. Secondly, we check each of the duplicated computations for equality. Upon detection of an inequality the corresponding computations have to be repeated. The check mechanism is implemented with only seven assembler instructions, which load the two computed values, compare them and jump according to the result. In this way, we reduce the performance overhead and, even more important, reduce the probability of a fault hitting the checker itself.

As a more sophisticated and more efficient approach, an application engineer could use the variable ranking together with his expert knowledge to create an application-specific protection scheme.

IV. 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. For robust fusion of the measurement data of the sensors the IMU is equipped with, we apply the Bayesian filtering scheme [7]. In particular, we recursively estimate the posterior probability density

$$p(\mathbf{x}_t \mid \mathbf{z}_{1:t}, \mathbf{u}_{1:t}) \tag{1}$$

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.

In our application, we represent the three-dimensional orientation by a unit quaternion. The predictive data are the measurements of the three-dimensional rotational velocity $\mathbf{u}$ of the gyroscope. The corrective measurements are that of the accelerometer and the magnetometer similar to [19]. We assume an application for orientation estimation in which the translational accelerations of the IMU are low compared to the gravity and can be neglected. Under this assumption, the acceleration measurements correspond to the gravity vector and can be used in a straight forward way to correct the estimated tilt of the IMU. In addition, the heading estimate is corrected using the horizontal projection of the magnetic field measurements, which correspond to the earth's magnetic field.

The Bayesian filtering scheme can be implemented in several different ways [7]. In the following, we will describe the extended Kalman filter and the particle filter.

A. Extended Kalman Filter

The extended Kalman filter (EKF) [20] is an efficient implementation of the Bayes filter which assumes Gaussian distributed uncertainty. It linearizes the system dynamics in a first order Taylor expansion and applies the recursive Kalman filter update, which exactly computes the state estimate of Gaussian linear systems. At time t the EKF represents the posterior $p(\mathbf{x}_t \mid \mathbf{z}_{1:t}, \mathbf{u}_{1:t})$ with a Gaussian $\mathcal{N}(\mathbf{x}_t; \boldsymbol{\mu}_t, \Sigma_t)$ with mean $\boldsymbol{\mu}_t$ and covariance Σ_t.

B. Particle Filter

The particle filter is a sample based implementation of the Bayes filter [21]. Here, the posterior is approximated by a set $\mathcal{M} = \left\{\left(\mathbf{x}^{[i]}, w^{[i]}\right) \mid i \in [1, N]\right\}$ of weighted particles, where each particle corresponds to a possible state $\mathbf{x}^{[i]}$ and has an assigned weight $w^{[i]}$. Each recursive update of the posterior is performed in three steps. Firstly, each particle is propagated from $\mathbf{x}_{t-1}^{[i]}$ to $\mathbf{x}_t^{[i]}$ using a sampled noise value on the predictive data $\mathbf{u}_t$ in the *prediction step*. Secondly, in the *correction step*, each particle is weighted according to the new corrective measurement $\mathbf{z}_t$. In the *resampling step*, a new generation of particles is drawn from $\mathcal{M}$ (with replacement) such that each sample in $\mathcal{M}$ is selected with

a probability that is proportional to its weight. After each update cycle, the state estimate $\boldsymbol{\mu}_t$ is computed as the weighted mean of all particles.

C. Filter Evaluation

We evaluate the accuracy of the filter algorithms by comparing the orientation estimates $\boldsymbol{\mu}_t$ to the actual orientations ("ground truth") $\mathbf{x}_t^\star$ at all time steps. Therefore, we evaluate the root mean square error (RMSE)

$$\mathrm{RMSE} = \sqrt{\frac{1}{T}\sum_{t=1}^{T}\left(\measuredangle\left(\boldsymbol{\mu}_t, \mathbf{x}_t^\star\right)\right)^2} \tag{2}$$

of the orientation estimates, where $\measuredangle(\cdot,\cdot)$ is the angular distance between two quaternions.

V. Experimental Results

A. Experimental Setup

In our experiments we use the ourMIPS processor, which runs on Altera Cyclone II FPGA starter boards [22]. The FPGA boards are connected to the PC via serial communication. We consider the three-dimensional orientation estimation using IMU data as software application under test. We examined two filters for sensor data fusion, namely the EKF and the particle filter, as described in Section IV. The input data was collected with an IMU mounted on 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 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 into one of two specific registers of the register file. Fault injection experiments showed, that faults in these registers produce many calculation errors. This gives us the required data for our analysis as described in Section III-B. In a second step, we implement the checker mechanism (Section III-C) 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 is indicated by many system crashes due to injected faults in these registers. The checker mechanism serves as a validation of the previous step.

B. Critical Variables

Figure 3 shows two rankings of critical variables for the EKF. Both rankings were created from the same set of experimental runs. Taking into account only the runs with critical errors is also a reasonable view, 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. Here, it is interesting to note, that the variable "pitch" from function "correction_mag" is listed on the second place when counting all errors, but does not appear when

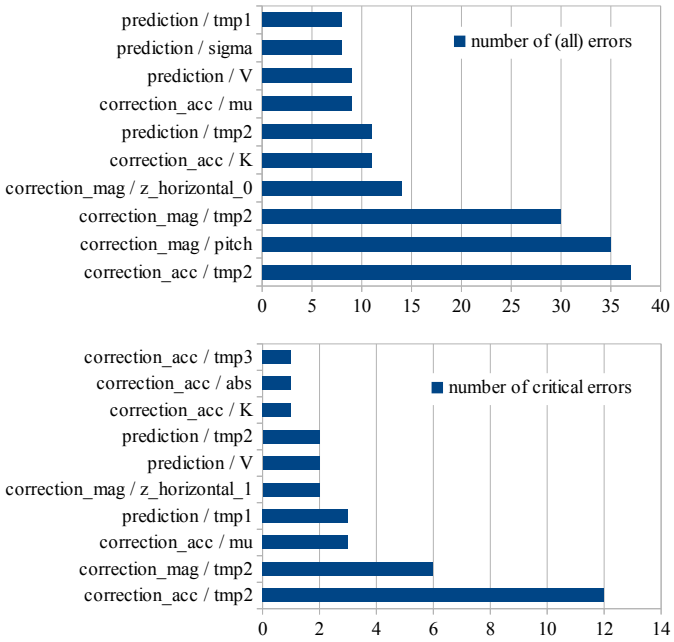

Figure 3. The critical variable statistics created counting all errors (top) and only the critical errors (bottom) of the EKF in 5000 runs. Each bar is labeled with the function the variable appears in and the variable itself.

counting only the critical errors. The reason for this is, that on the one hand "pitch" consumes a lot of computation time and therefore is hit by a fault with a high probability. On the other hand it has only a small influence on the overall result of the algorithm and thus only causes small computation errors. Consequently, "pitch" should not be protected.

In our experiments we also evaluated the particle filter for identifying its critical variables. The results indicate 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. Most of the computation time is spent for the calculation of the particle values. In case of calculation errors in particle values a low weight is assigned to the particle in the correction step and it eventually dies out during a resampling step. For that reason the particle filter 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 4 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

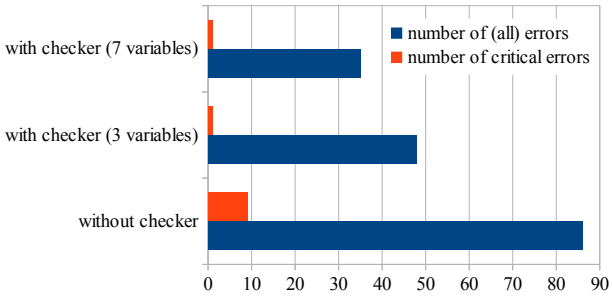

Figure 4. Comparison of different protection levels of the EKF application in 7000 runs.

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 3). 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 of the number or 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.

VI. 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.

In the future we plan to further extend our framework. Firstly, we intend to identify also critical variables with respect to system crashes. Secondly, we want to give a user more control over the fault injection for evaluating an application more efficiently. For this purpose we will allow to select only a part of the source code for fault injection.

Acknowledgements

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

References

[1] M. Porter, J. Wilkinson, K. Walsh, B. Sierawski, K. Warren, R. Reed, and G. Vizkelethy, "Soft error reliability improvements for implantable medical devices," in *International Reliability Physics Symposium*, pp. 488–491, 2008.

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

[3] 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.

[4] 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.

[5] G. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. August, "SWIFT: Software implemented fault tolerance," in *International Symposium on Code Generation and Optimization*, pp. 243–254, 2005.

[6] 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.

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

[8] R. Velazco, R. Ecoffet, and F. Faure, "How to characterize the problem of SEU in processors & representative errors observed on flight," in *International On-Line Testing Symposium*, pp. 303–308, 2005.

[9] L. Sterpone, D. Sabena, S. Campagna, and M. S. Reorda, "Fault injection analysis of transient faults in clustered VLIW processors," in *International Symposium on Design and Diagnostics of Eletronic Circuits & Systems*, pp. 207–212, 2011.

[10] J. Sartori, J. Sloan, and R. Kumar, "Stochastic computing: Embracing errors in architecture and design of processors and applications," in *International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pp. 135–144, 2011.

[11] S. A. Seshia, W. Li, and S. Mitra, "Verification-guided soft error resilience," in *Design, Automation & Test in Europe Conference & Exhibition*, pp. 1–6, 2007.

[12] S. Bergaoui, P. Vanhauwaert, and R. Leveugle, "A new critical variable analysis in processor-based systems," in *Transactions on Nuclear Science*, pp. 1992–1999, 2010.

[13] N. Nakka, K. Pattabiraman, and R. Iyer, "Processor-level selective replication," in *International Conference on Dependable Systems and Networks*, pp. 544–553, 2007.

[14] M. Portela-Garcia, A. Lindoso, L. Entrena, M. Garcia-Valderas, C. Lopez-Ongil, B. Pianta, L. B. Poehls, and F. Vargas, "Using an FPGA-based fault injection technique to evaluate software robustness under SEEs: A case study," in *Latin American Test Workshop*, pp. 1–6, 2011.

[15] 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.

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

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

[18] 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.

[19] A. Sabatani, "Quaternion-based extended kalman filter for determining orientation by inertial and magnetic sensing," *IEEE Transactions on Biomedical Engineering*, vol. 53, no. 7, pp. 1346–1356, 2006.

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

[21] 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.

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