An FPGA-Based Framework for Run-time Injection and Analysis of Soft Errors in Microprocessors

M. Sauer[1], V. Tomashevich[2], J. Müller[1], M. Lewis[1], A. Spilla[1], I. Polian[2], B. Becker[1], and W. Burgard[1]

[1]Albert-Ludwigs-University
Georges-Köhler-Allee 51
79110 Freiburg i. Br., Germany
{sauerm,muellerj,spilla,lewis,becker,burgard}@informatik.uni-freiburg.de

[2]University of Passau
Innstraße 43
94032 Passau, Germany
{victor.tomashevich,ilia.polian}@uni-passau.de

Abstract—**State-of-the-art cyber-physical systems are increasingly deployed in harsh environments with non-negligible soft error rates, such as aviation or search-and-rescue missions. State-of-the-art nanoscale manufacturing technologies are more vulnerable to soft errors. In this paper, we present an FPGA-based framework for injecting soft errors into user-specified memory elements of an entire microprocessor (MIPS32) running application software. While the framework is applicable to arbitrary software, we demonstrate its usage by characterizing soft errors effects on several software filters used in aviation for probabilistic sensor data fusion.**

I. Introduction

Soft errors cause nodes within a circuit to temporarily fail. They are typically generated by ionizing radiation from α-particles or cosmic rays [10]. As modern transistors shrink, the probability of a fault occurring increases. Soft errors have traditionally been a concern in safety-critical systems including medical devices [4] and aviation/space applications where chips operate under increased radiation [7], [23]. Today, cost pressure and energy constraints limit the applicability of massive redundancy, while increased complexity of calculations performed in novel applications such as robots on search-and-rescue missions necessitate the usage of powerful microprocessors.

On the positive side, there is also emerging evidence that many applications, including image-processing [24] and artificial-intelligence algorithms [18], are *resilient*, i.e., produce tolerable results even when they are affected by soft errors during operation. Before spending significant hardware resources for radiation hardening or redundancy, it is necessary to understand which impact soft errors would have on the target application.

In this paper, we introduce an FPGA-based fault-injection platform to test and simulate transient faults in microprocessors using FPGAs. The platform can provide insight into the design's soft error characteristics, allowing a designer to logically harden a chip by adding error correction and test the improvements before the chip is actually produced. The platform is flexible with respect to the target processor (which is synthesized on the FPGA and equipped with a scan-based fault injector), the application software, the input data, and the profiles of faults injected. The *generic fault-injection manager*, running on the PC side, encapsulates all the technical details and controls the FPGA over a communication protocol (transfers fault-injection information and the input data to the FPGA and receives and evaluates the obtained output data). By allowing the FPGA to communicate with further external devices, we can test the susceptibility of the processor when it is running its native applications in its usual environment.

Although a significant amount of research has resulted in software simulation methods [1], [5], [22], [11], these tools are not powerful enough to simulate entire SoCs running real applications such as the probabilistic filters reported here. Radiation testing [27], [29] can provide insight but requires access to a radiation testing facility and non-trivial conversion of the error rates observed [16]. Therefore, FPGA-based emulation has been used to replace [6], [19] or complement [17], [25] software simulation. A good overview of much of this research can be found in [12]. More recently, the partial-reconfiguration features of FPGAs for fault injection [2], [21] and large parallel emulators al. [8] have been utilized. Ongil et al. [20] discuss various types of fault injection techniques for medium-size non-programmable circuits. In [14], these techniques are applied to the Leon 2 processor and compared with the software-based code-emulated upsets method [13].

Our objective is to use an entire System on Programmable Chip (SoPC) design that incorporates a MIPS32 based microprocessor with peripheral devices such as general-purpose I/Os and serial UARTs, along with a programmable fault injector. With respect to the software approaches, we are still able to simulate millions of clock cycles per second in real time for our SoPC, and our approach does not limit what areas of the processor faults can be injected into. In our case, all storage elements can be affected by the transient faults that we inject. While we use ideas from some of these publications (most notably the general design of the fault injector from [6]), our framework focuses on evaluating complex software applications.

The FPGA emulation architecture is described in Section II. Background on Bayesian filters used as the software application on the test is given in Section III. Section IV presents experimental results. Section V concludes the paper.

II. Run-time Fault Injection

Our FPGA-based fault-injection framework, shown in Figure 1, consists of an FPGA part on which the actual fault-injection experiment is run and the PC part. The latter is used to (1) communicate inputs of the experiment to the FPGA part; (2) control the experiment execution, and (3) receive and analyze its outcomes. The FPGA part implements the target processor on which the application under test is run; the fault injector; the memory; and controllers to handle input/output

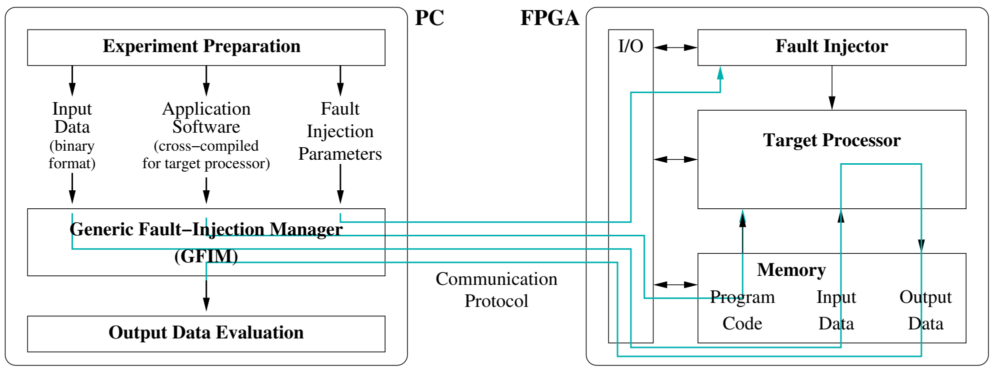
PC
FPGA
Experiment Preparation
Input Data (binary format)
Application Software (cross–compiled for target processor)
Fault Injection Parameters
Generic Fault–Injection Manager (GFIM)
Output Data Evaluation
Communication Protocol
I/O
Fault Injector
Target Processor
Memory
Program Code
Input Data
Output Data

Fig. 1. Hardware platform and architecture of our run-time fault injection system. The blue arrows indicate the flow of information.

and further peripherals. The 32-bit *OurMIPS* processor [3] used in our experiments can execute all MIPSv1 instructions, supports hardware interrupts. Floating point arithmetic is done through software emulation. The processor's general-purpose I/O (GPIO) ports can be connected to sensors or other devices.

The fault-injection framework has been designed with a view to provide maximal modularity and well-defined interfaces between experiment steps, thus improving flexibility. This is achieved by using a generic, application-independent software module called *generic fault-injection manager* (GFIM). It takes the input data, the application software, and the fault injection parameters in a well-defined format, and hands them over to the FPGA part over a specified communication protocol. It is possible to designate the individual flip-flops where the faults will be injected along with injection times, or to specify an error rate according to which the faults will be randomly injected. Any of GFIM's inputs can be exchanged with no need to modify the GFIM itself or the FPGA part.

The actual fault-injection experiment is executed on the FPGA (controlled by the GFIM). The *Fault Injector* can inject transient faults into the OurMIPS processor using the shadow-scan mechanism (similar to [6]) shown in Figure 2. The fault injection information is prepared using shift operations while not disturbing the normal execution of the running software. Additional injection sites within logic blocks could be included thereby increasing the transient site fault coverage. We also added logic to prevent the fault injector from being reconfigured by the processor once an application was started. This could for instance happen if a previous transient fault caused the processor to jump to some uninitialized memory location, resulting in the core executing random instructions.

The output data is written into the processor's memory on the FPGA. Finally, the collected results are communicated to GFIM over the same communication protocol. GFIM does not perform any result evaluation itself but forwards the received data to a user-defined output data evaluation procedure. The application developer can perform comprehensive tests without having to care about the FPGA communication, the fault-injection mechanism or other hardware-related issues. In case of probabilistic filters considered here, this procedure evaluates the magnitude of the result deviation observed using a special metric RMSE described below and also detects outliers.

Some errors result in the target processor crashing or becoming totally unresponsive. We refer to such errors as *total system faults* (TSFs) and identify them using a timeout, after which the next session is started. Note that TSFs can be detected in operation using a watchdog processor. Our framework provides a versatile tool to learn which errors tend to result in TSFs.

III. Probabilistic Sensor Data Fusion

Throughout the experiments presented in this paper, we consider the problem of estimating the position x of a mobile robot in a known environment with one degree of freedom moving back and forth, e.g., in a delivery task between two stations. The robot is equipped with a distance sensor which continuously takes noisy distance measurements from its current position to one station and receives a velocity command in each time step. We maintain the probability density function $p(x_t \mid z_{1:t}, u_{1:t})$ of the position x_t of the robot at time t given all the sensor data $z_{1:t}$ and the control inputs $u_{1:t}$ up to time t. This probability is calculated recursively using the Bayesian filtering scheme [28]:

$$p(x_t \mid z_{1:t}, u_{1:t}) = \eta_t \cdot p(z_t \mid x_t) \cdot \int p(x_t \mid u_t, x_{t-1})\, p(x_{t-1} \mid z_{1:t-1}, u_{1:t-1})\, dx_{t-1}\ , \qquad (1)$$

where η_t is a normalizer. The term $p(x_t \mid u_t, x_{t-1})$ is the state transition probability of the motion model, and $p(z_t \mid x_t)$ is the measurement probability of the sensor model.

The motion and measurements of the robot are modeled by a linear function with Gaussian distributed noise: the motion model is distributed according to $p(x_t \mid u_t, x_{t-1}) = \mathcal{N}(x_{t-1} + \Delta t\, u_t, R_t)$, where Δt is the time elapsed since the last step of the filter. The sensor measures the distance to the base station located at $x = 0$ and therefore its measurements follow the $p(z_t \mid x_t) = \mathcal{N}(x_t, Q_t)$ probability distribution. The covariances R_t and Q_t can be obtained in a straight forward manner by comparing recorded data to ground-truth data. In the following, we describe two implementations of the Bayesian filter, namely the Kalman filter and the particle filter, which are widely used for state estimation in robotics and many other fields.

1) Kalman Filter: The Kalman filter [15] assumes linear system dynamics with Gaussian distributed noise and exactly computes the state estimate of such systems. At time t the

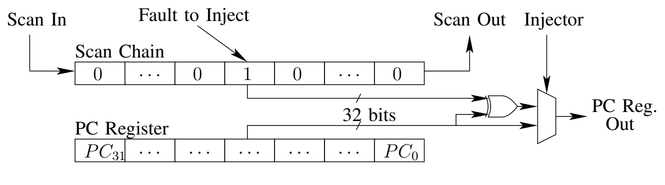

Fig. 2. PC Register with scan chain and fault injector.

Table 1. Root mean square errors (RMSE [m]) and the numbers of total system failures (TSF, per 100 sessions) for different filters and fault rates.

	No faults	10 faults/s		50 faults/s		100 faults/s	
Type	RMSE	RMSE	TSF	RMSE	TSF	RMSE	TSF
Kalman filter (2471 samples)							
float	0.021	0.021	6	2.8E35	17	0.021	18
fx32[4].[28]	0.021	0.021	5	0.048	15	0.030	24
fx32[13].[19]	0.021	0.021	10	17.451	12	0.766	21
fx16[4].[12]	0.021	0.021	4	0.032	10	0.026	11
fx16[13].[3]	0.142	1.122	5	0.147	16	41.398	15
Kalman filter (precalculated covariances, 2471 samples)							
float	0.021	0.021	9	0.022	15	13.7E14	23
fx32[4].[28]	0.021	0.021	11	0.023	13	0.021	14
fx32[13].[19]	0.021	0.021	5	1.734	10	3.082	20
fx16[4].[12]	0.021	0.022	3	0.023	8	0.022	8
fx16[13].[3]	0.147	0.156	1	2.914	9	0.479	12
Particle filter (10 particles, 21 samples)							
float	0.023	0.023	2	1.15E17	9	0.034	9
Particle filter (50 particles, 21 samples)							
float	0.021	0.021	8	720.299	22	4.3E35	15

Table 2. Numbers of sessions resulting in out-of-range values (OOR), and RMSE excluding these sessions.

	No faults	10 faults/s		50 faults/s		100 faults/s	
Type	RMSE	RMSE	OOR	RMSE	OOR	RMSE	OOR
Kalman filter (2471 samples)							
float	0.021	0.021	0	0.022	1	0.021	0
fx32[4].[28]	0.021	0.021	0	0.048	0	0.030	0
fx32[13].[19]	0.021	0.021	0	0.021	1	0.032	2
fx16[4].[12]	0.021	0.021	0	0.032	0	0.026	0
fx16[13].[3]	0.142	0.147	1	0.147	0	0.146	1
Kalman filter (precalculated covariances, 2471 samples)							
float	0.021	0.021	0	0.022	0	0.033	1
fx32[4].[28]	0.021	0.021	0	0.022	0	0.021	0
fx32[13].[19]	0.021	0.021	0	0.024	2	0.021	1
fx16[4].[12]	0.021	0.022	0	0.023	0	0.022	0
fx16[13].[3]	0.142	0.156	0	0.147	2	0.159	1
Particle filter (10 particles, 21 samples)							
float	0.023	0.023	0	0.023	1	0.034	0
Particle filter (50 particles, 21 samples)							
float	0.021	0.021	0	0.021	1	0.024	2

Kalman filter represents the belief $p(x_t \mid z_{1:t}, u_{1:t})$ with a Gaussian $\mathcal{N}(\mu_t, \Sigma_t)$ with mean μ_t and covariance Σ_t. The linear motion model is $x_t = A_t\, x_{t-1} + B_t\, u_t + \varepsilon_t$ and the sensor model is $z_t = C_t\, x_t + \delta_t$ where $\varepsilon_t \sim \mathcal{N}(0, R_t)$ and $\delta_t \sim \mathcal{N}(0, Q_t)$ are random noise variables. Algorithm 1 depicts one step of the Kalman filter fusing a control input and a sensor measurement into the belief of the filter.

Algorithm 1 Kalman filter step

```
Input: $\mu_{t-1}$, $\Sigma_{t-1}$, $u_t$, $z_t$
Output: $\mu_t$, $\Sigma_t$
    $\bar{\mu}_t = A_t\, \mu_{t-1} + B_t\, u_t$
    $\bar{\Sigma}_t = A_t\, \Sigma_{t-1}\, A_t^T + R_t$
    $K_t = \bar{\Sigma}_t\, C_t^T\, (C_t\, \bar{\Sigma}_t\, C_t^T + Q_t)^{-1}$
    $\mu_t = \bar{\mu}_t + K_t\, (z_t - C_t\, \bar{\mu}_t)$
    $\Sigma_t = (I - K_t\, C_t)\, \bar{\Sigma}_t$
```

In our application, the linear motion coefficients are $A_t = 1$ and $B_t = \Delta t$ and, the measurement coefficient is $C_t = 1$. As we assume the motion and measurement covariance R and Q to be independent of t, we optionally can precalculate the Kalman gains K_t and the covariances Σ_t in advance for all t and store them in a lookup table to improve efficiency.

2) Particle Filter: The particle filter is a sample based approach to state estimation commonly known as Monte Carlo localization [9]. The current belief is approximated by a set $\mathcal{M} = \{\langle x^{[i]}, w^{[i]}\rangle \mid i \in [1, N]\}$ of weighted particles, where each particle corresponds to a possible robot position and has an assigned weight $w^{[i]}$. The belief update from (1) is performed according to the following three alternating steps:

1) In the *prediction step*, we propagate each particle by drawing a successor position from the motion model $p(x_t^{[i]} \mid u_t, x_{t-1}^{[i]})$ given the control u_t.
2) In the *correction step*, we integrate a new measurement z_t by assigning a new weight $w^{[i]}$ to each particle according to the sensor model $p(z_t \mid x_t^{[i]})$.
3) In the *resampling step*, we draw a new generation of particles 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, we compute the position estimate μ_t as the weighted mean of all particles.

3) Filter Evaluation: We evaluate the performance of the filter algorithms by comparing the position estimates μ_t to the actual positions ("ground truth") $x_t^\star$ at all time steps using the root mean square error (RMSE)

$$\text{RMSE} = \sqrt{\frac{1}{T} \sum_{t=1}^{T} (\mu_t - x_t^\star)^2}\ . \tag{2}$$

IV. Experimental Results

We implemented the hardware part of the fault-injection framework using a Cyclone II FPGA Starter Board. The entire design uses approximately a third of the FPGA (Altera EP2C20), and our SoPC runs at 12.5 MHz. We considered one-dimensional Bayesian filters (the regular Kalman filter, a version of the Kalman filter with precalculated covariance matrices and Kalman gains, and the particle filter) as software applications under test. The input data was generated from localizing a MobileRobots Pioneer 3-DX wheeled robot while traveling back and forth between two walls. A forward-pointing Devantech SRF10 miniature sonar sensor mounted on the robot and dead reckoning odometry (i.e. position based on the rotation of the wheels) were the data sources. The ground truth position (i.e. the actual physical positions of the robot) was obtained using a SICK LMS 291 laser range finder, also mounted on the robot. All data was prerecorded and stored on the hard disk of the PC part of the fault-injection platform before the experiments were run.

Table 1 summarizes the results of the filters using the standard IEEE 32-bit floating-point numbers ("float") and 32-bit and 16-bit fixed-point numbers with m magnitude bits and f fractional bits ("fx32[m].[f]" and "fx16[m].[f]", respectively). For instance, fx32[4].[28] is the 32-bit fixed-point representation with 4 bits allocated for the integer part. All data generated by the sensor had a maximum range of about 2 meters with an accuracy of about one centimeter, so at least 3 magnitude bits and 7 fractional bits are required. In particular, fx[13].[3] is insufficient to accurately represent all the fractional bits, which is also reflected by elevated fault-free RMSE. For each filter/number type considered, we performed one experiment without injecting any faults and three fault-injection experiments, each of which consists of 100 sessions with error rates of 10, 50 and 100 faults per second, respectively. One fault-injection session took around 1.5 seconds for the Kalman filters and around 10 seconds for the more complex particle filter. This corresponds to millions of instructions and would be very time consuming if simulation would be performed in software.

We report the root mean square error (RMSE, see (2)) to demonstrate the quality deterioration of the filter, and the number of total system failures (TSF) to assess the stability of the algorithm under soft errors. The RMSE numbers are averaged over $(100-\mathrm{TSF})$ sessions for which results are available. If most-significant bits or exponent bits of floating-point numbers are flipped by fault injection, the RMSE can assume very large values exceeding by far the robot's maximum range of 2 meters. One such out-of-range value is often enough to produce an out-of-range average result. For example, the value of 13.7E14 in the row "float" of Table 1 is caused by an out-of-range result in just one session. Table 2 shows the average RMSEs obtained when all the sessions which produced RMSE values over 5 meters are excluded (the number of such sessions is quoted in columns "OOR").

It can be seen that errors of largest magnitude occur when float numbers are used. For fixed-point numbers, more magnitude bits result in a higher significance of some of the bits and, as a consequence, larger errors. It appears that not using more magnitude bits than are necessary from the application's point of view is the strategy which minimizes the average error. The RMSE tends to grow when more errors are injected. When increasing the number of injected transient faults, the system crashes (experiences a TSF) significantly more often.

The number of TSFs for the Kalman filter with precalculated covariances tends to be less than for its counterpart without precalculation, which is due to its slightly lower run times and therefore less faults injected per session. Similarly, calculations using fx16 are faster than using fx32 and result in a lower number of TSFs. Using this information, and looking deeper at each test case would provide a designer with more insight on which parts of the processor should be hardened to make the system more robust against transient faults.

V. Conclusion

We presented an FPGA-based fault-injection framework to test and simulate transient faults in arbitrary software applications on a common 32-bit processor. Our architecture encapsulates technical details in a generic fault injection manager module and overcomes memory limitations of earlier methods by an efficient distribution of test data between PC, FPGA and external memory. The platform enables us to seamlessly connect our SoPC to external devices and therefore can execute real applications in their usual environments. Using our framework, the designer can test both the hardware and software aspects of a system before silicon is available.

Our system allowed us to experimentally evaluate the vulnerability of Bayesian filter localization algorithms under different representations of real numbers. In the future, we plan to test other types of software applications and study the behavior of systems combining hardware redundancy and software-implemented fault tolerance [26] to design highly dependable systems in an energy- and cost-efficient manner.

References

[1] J. Aidemark et al. Generic object-oriented fault injection tool. In *Int'l Conf. on Dependable Systems and Networks*, pages 83–88, 2001.

[2] L. Antoni, R. Leveugle, and B. Feher. Using run-time reconfiguration for fault injection in hardware prototypes. In *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 245–252, 2002.

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

[4] P. Bradley and E. Normand. Single event upsets in implantable cardioverter defibrillators. In *IEEE Transactions on Nuclear Science*, pages 2929–2940, 1998.

[5] J. Carreira, H. Madeira, and J. Silva. Xception: a technique for the experimental evaluation of dependability in modern computers. *IEEE Transactions on Software Engineering*, pages 125–136, 1998.

[6] P. Civera, L. Macchiarulo, M. Rebaudengo, M. S. Reorda, and M. Violante. An FPGA-based approach for speeding-up fault injection campaigns on safety-critical circuits. In *Journal of Electronic Testing*, pages 261–271, 2002.

[7] J. Clark and D. Pradhan. Fault injection: a method for validating computer-system dependability. *Computer*, pages 47–56, 1995.

[8] J.-M. Daveau et al. An industrial fault injection platform for soft-error dependability analysis and hardening of complex system-on-a-chip. In *IEEE Int'l Reliability Physics Symp.*, pages 212–220, 2009.

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

[10] P. Dodd and L. Massengill. Basic mechanisms and modeling of single-event upset in digital microelectronics. In *IEEE Transactions on Nuclear Science*, pages 583–602, 2003.

[11] E. Dupont, M. Nicolaidis, and P. Rohr. Embedded robustness IPs for transient-error-free ICs. *Design & Test of Comp.*, pages 56–70, 2002.

[12] B. F. Dutton. Embedded soft-core processor-based built-in self-test of field programmable gate arrays. Master's thesis, Auburn Univ., 2010.

[13] F. Faure et al. Impact of data cache memory on the single event upset-induced error rate of microprocessors. *IEEE Transactions on Nuclear Science*, pages 2101–2106, 2007.

[14] M. Garcia-Valderas et al. Two complementary approaches for studying the effects of SEUs on digital processors. *IEEE Transactions on Nuclear Science*, pages 924–928, 2007.

[15] R. E. Kalman. A new approach to linear filtering and prediction problems. *ASME-Journal of Basic Engineering*, March(82):35–45, 1960.

[16] H. Kobayashi et al. Comparison between neutron-induced system-SER and accelerated-SER in SRAMs. In *Int'l Reliability Physics Symp.*, pages 288–294, 2004.

[17] M. Kochte, R. Baranowski, and H.-J. Wunderlich. Zur Zuverlässigkeitsmodellierung von Hardware-Software-Systemen. In *GMM/GI/ITG-Fachtagung Zuverlässigkeit und Entwurf*, 2008.

[18] X. Li and D. Yeung. Application-level correctness and its impact on fault tolerance. In *Int'l Symp. on High Performance Computer Architecture*, pages 181–192, 2007.

[19] F. Lima, S. Rezgui, L. Carro, R. Velazco, and R. Reis. On the use of VHDL simulation and emulation to derive error rate. In *Radiation Effects on Components and Systems Conference*, pages 253–260, 2001.

[20] C. Lopez-Ongil et al. Autonomous fault emualtion: A new FPGA-based acceleration system for hardness evaluation. *IEEE Transactions on Nuclear Science*, pages 252–261, 2007.

[21] C. Lopez-Ongil et al. A unified environment for fault injection at any design level based on emulation. *IEEE Transactions on Nuclear Science*, pages 946–949, 2007.

[22] B. Nicolescu and R. Velazco. Detecting soft errors by a purely software approach: Method, tools and experimental results. In *Design, Automation, and Test in Europe Conference*, 2003.

[23] E. Normand. Single-event effects in avionics. In *IEEE Transactions on Nuclear Science*, pages 461–474, 1996.

[24] D. Nowroth, I. Polian, and B. Becker. A study of cognitive resilience in a JPEG compressor. In *Int'l Conf. on Dependable Systems and Networks*, pages 32–41, 2008.

[25] M. Rebaudengo et al. Combined software and hardware techniques for the design of reliable IP processors. In *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 265–273, 2006.

[26] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. Swift: Software implemented fault tolerance. In *International Symposium on Code Generation and Optimization*, pages 243–254, 2005.

[27] N. Seifert, X. Zhu, and L. Massengill. Impact of scaling on soft-error rates in commercial microprocessors. In *IEEE Transactions on Nuclear Science*, pages 3100–3106, 2002.

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

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