Run-time Soft Error Injection and Testing of a Microprocessor using FPGAs

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

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

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

***Abstract*—Nowadays, soft errors in logic circuits are becoming increasingly important. This is especially true in the ever shrinking nanometer technologies, and aerospace applications where soft errors are more prevalent. Previous soft error injection simulation methods using FPGAs have usually been limited to small test circuits such as ISCAS89, or parts of a larger circuit. Other software approaches have been proposed that simulate soft errors by adding extra instructions that the processor must execute. In this paper, we introduce our FPGA based transient fault injection system that can handle all memory elements of an entire microprocessor (MIPS32) while connected and running within a complete system. This allows us to perform fault injection and analysis when the system is live and actually running arbitrary applications on the processor. We test our implementation, and the effects that soft errors have on a software filter used in aviation for probabilistic sensor data fusion, as this algorithm would be run at higher altitudes where soft errors are more frequent. Our experimental results not only show that our method is extremely fast and versatile, but that it also allows us to test how software applications perform under a wide range of fault conditions.**

I. Introduction

Transient faults cause nodes within a circuit to temporarily fail. The source of these types of faults is typically due to ionizing radiation from α-particles or cosmic rays [14]. Since external natural events cause transient faults to occur, they are not reproducible (i.e. the fault may occur during one test and not in a second identical test). The probability of a single node upset is normally relatively small. However, as modern transistors shrink, the probability of a transient fault occurring increases. For certain fields like medical devices, where proper operation must be guaranteed, transient faults have been studied for quite some time [9]. Furthermore, in aviation/space applications where chips operate under increased radiation, these faults can be problematic [13], [23].

To rate how susceptible a chip is to a soft error, terms like *mean time to failure* (MTTF) or *failure in time* (FIT) are used. The FIT rate is defined as the number of failures per billion hours of operation, i.e. 114,000 years. A typical FIT rate of a commercial semiconductor circuit is between 1 and 100. Consequently, MTTF is between 1,140 and 114,000 years. Therefore, acceleration techniques are needed to measure MTTF [26]. To accelerate soft error testing, increased levels of radiation are used. The results are then scaled down for normal levels of radiation. However, it is difficult to validate the scaling factors (calibration) [19], thus, failure rates are often gross estimates. This has motivated much of the previous simulation work discussed in Section II.

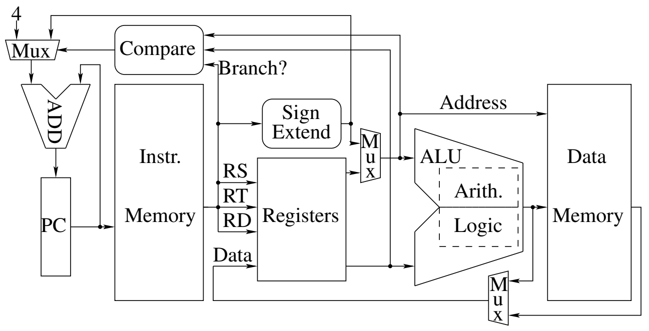

Fig. 1. OurMIPS: Processor from [8] and used in courses taught at the University of Freiburg.

Additionally, during the development of a chip, a radiation testing facility is not always available. For a hardware designer, the ability to test how susceptible a design is to transient faults can be of great importance. Accelerated simulation techniques can provide a designer with the opportunity to test a design, providing insight into its soft error characteristics. Furthermore, these techniques can be used during the developmental stages of a design, allowing a designer to logically harden a chip (i.e. by adding error correcting elements) and test the improvements before the chip is actually produced. This can result in increased product quality, while also reducing costs associated with overall product development and testing.

The work presented in this paper tries to address these issues by providing a platform to test and simulate transient faults in microprocessors using FPGAs. The flexibility of an FPGA not only allows us to model many different System on Programmable Chip (SoPC) systems, but to test these systems at a significantly faster rate than would be possible using software. With the addition of our automated PC based application which allows the user to easily configure which tests to perform, while also providing and generating a complete summary (including graphs) of the results, we are providing a platform that could be seamlessly integrated into a normal design flow. Furthermore, by allowing the FPGA to communicate with the external environment, we can test the susceptibility of the processor when it is running its native applications in its normal environment.

Before covering our platform in more detail, the terminology and previous work will be discussed next. Our emulation architecture and the software application that was tested is then described in Section III. Section IV presents our first experimental results, and Section V concludes the paper.

II. Preliminaries

A. Terminology and Motivation

In literature, the terms "transient fault" and "soft error" have been used interchangeably which can cause some confusion. Here, we define a transient fault as a single or multiple node upset directly attributable to excess charge carriers induced by external radiation (the terms single event upset (SEU) and single event transient (SET) are used in the Nuclear Science literature [7]). By contrast, we define a soft error as the impact of a transient fault that may persist beyond one clock cycle. For the impact to transfer from one clock phase to another, it must be captured in a storage element such as a latch or register. Thus, soft errors may manifest away from their originating location. Therefore, transient faults and soft errors can be viewed as cause and effect.

Strategies for addressing soft errors include the estimation of the vulnerability of a given device to these errors, and the design of soft error detection and correction circuitry. For each of these topics, it is useful to know how the circuit behaves in the presence of transient faults. For this purpose, software simulation methods for soft errors exist (including commercial tools such as IROC's Roban [15]). However, some specific characteristics of soft errors impose limits on the applicability and scalability of these software solutions. For instance, software injected transient faults can only affect certain parts of a processor, whereas soft errors can occur anywhere within a processor. Furthermore, software simulation methods are not powerful enough to simulate entire SoCs running real applications. Also, this work only considers logic based transient faults which are the most common type. α-particles and cosmic rays can also affect power supplies and system clocks causing extra signal delays and timing issues. However, these topics are beyond the scope of this paper.

B. Previous Work

A significant amount of research has been carried out on examining the real effects of soft errors [29], and on the ability to detect and correct them if they do happen [6], [17]. However, here we focus on the simulation and test aspects of transient faults and soft errors. In this context, [11] and [12] used a hardware FPGA approach to speedup the simulation process. Software emulation of transient faults, in which errors are inserted by adding extra lines of software to a program has also been introduced in [5], [10], [22]. Combined software/hardware methods have also been looked at in [20], [24]. A good overview of much of this research can be found in [16]. Compared to our method, the previous hardware based methods only simulate small test circuits such as ISCAS89 under the presence of random inputs. Here, we use an entire System on Programmable Chip (SoPC) design that incorporates a MIPS32 based microprocessor (see Figures 1 and 2). With respects to the software approaches, we are still able to simulate millions of clock cycles for our SoPC system per second, 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.

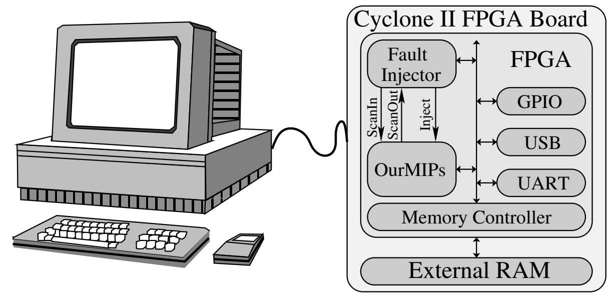

Fig. 2. Hardware platform and architecture.

In [21], an approach is presented that allowed the authors to emulate an 8051 micro-controller while injecting faults during run-time. However, their design was limited to a small 8 bit micro-controller because two entire copies of the system had to be implemented on the FPGA (i.e. a good implementation, and one suffering from transient faults). This further limited the memory available to run applications to a few kilobytes (using external memory, we have access to MB of space in our design). Also, the fault injector in [21] was based on a pseudo random number generator and was not programmable. Furthermore, they only considered soft errors, not transient faults. This can be inadequate as transient faults may propagate to multiple flip-flops, or due to logical, electrical or latching-window masking, they may not propagate to a single storage element [27].

III. Run-time Fault Injection

A. System Overview

The basic architecture flow for our FPGA based fault injection tool is shown in Figure 2. In our SoPC design, we used the MIPS32 instruction set compatible processor *OurMIPS* described in [8] and shown in Figure 1. This 32-bit processor can execute all MIPSv1 instructions, supports hardware interrupts, and floating point arithmetic can be done through software emulation. On average, it can execute one 32-bit instruction every two clock cycles. Through a common main bus, this processor is connected to the *Fault Injector*, external *Memory Controller*, *Serial UART*, *USB*, and multiple *GPIO* ports. The USB and Serial UART allow us to program the FPGA and communicate with the system while a software test program is executing. The GPIO ports can be connected to sensors, and/or other input/output/bidirectional devices. All components in our SoPC are memory mapped, allowing the processor to easily communicate with all parts of the system.

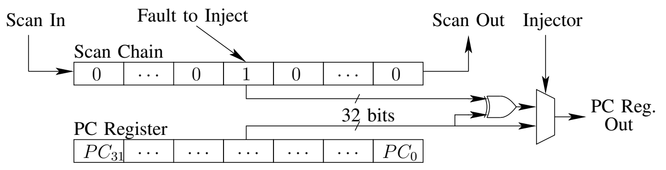

Fig. 3. PC Register with scan chain and fault injector.

Our fault injector (shown in Figure 2) can inject transient faults into the OurMIPS processor using the mechanism shown in Figure 3. Simply put, for every register or memory element in the OurMIPS processor, we have introduced a shadow register (top register in Figure 3 and similar to the approach taken in [11]). This allows us to shift in the fault location(s) while not disturbing the normal execution of the running software. This feature is required to allow the CPU to function correctly with devices such as the external memory and GPIO. If a normal scan chain was used for example, it could result in random data being written to the main SoPC bus when faults are being shifted in through the scan chain. Furthermore, if the processor was to be paused while the scan bits were shifted, this could break any timing requirements and/or protocols used to communicate with external devices over the GPIO ports. With our setup, only the simulated transient faults can affect the system's behavior.

Another advantage of our shadow scan chain is that it is XORed with the output of the real registers allowing the simulated transient faults to propagate to multiple registers, or in many cases to no registers. This can amplify or remove the effects of a transient fault from the system, and we consider this more realistic behavior as we briefly discussed in Section II-B[1]. Additionally, if the SoPC is not area limited, additional injection sites within logic blocks could be included thereby increasing the transient site fault coverage.

Regarding the fault injector, we designed a flexible, programmable, and full featured unit. Programmable in this sense means that before the processor executes the software application we are testing, it can program the fault injector with all the test information. For example, we can program how many faults or how often faults should occur. Furthermore, we can use randomly generated faults and time periods, or specify exactly when and where the faults should happen. This will allow us in the future to test specific parts of the processor to see how susceptible each part is to transient faults. Also, by specifying an initialization value for the random fault injector, we can reproduce the results of any previous test.

For each actual test, the injector has two modes of operation. The first mode is the programming mode that allows us to set all the parameters. The second mode is the running mode. Once switched into running mode, the registers in the fault injector can no longer be written to, and it will inject the specified faults as instructed. This was done to ensure that the fault injector was not 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.

[1]Through a simple modification (e.g. moving the shadow register and XOR gate before the real registers), we could simulate soft errors occurring only in memory elements as many of the previous works have done.

B. Work Flow

Using our system in Figure 2, the work flow for performing transient fault simulations is as follows. First, we program the FPGA with the SoPC described earlier in Section III-A. Again, this is unlike other work as we do not need to recompile our design for every test, we just need to program the FPGA which takes a few hundred milliseconds. This allows us to quickly run many small tests, and produce averages over a number of test runs which is important when using random transient fault injection to characterize the system under test.

Once the FPGA is configured, we send the initial setup configuration and parameters to the CPU which then configures the fault injector and other components. Next we send the software application to the CPU, which is then initially stored in the external memory. Once all this is complete, the fault injector is started and the CPU begins executing the test application. During the execution of our application we can monitor the processor's progress and status on the PC through the USB and serial connections. Partial results can also be sent over the serial cable and tested on the PC for correctness. For the results in Section V, we only monitored the programs execution at certain time points. However, to collect more precise information an embedded logic analyzer like Altera's "SignalTap II Logic Analyzer" could be used [2]. This allows one to monitor registers, signal lines, and most other internal components contained within the FPGA. This extra monitoring comes at the cost of FPGA area and possibly reduced frequency, but it shows the flexibility of our overall design. To summarize, our work flow contains the following six steps:

1) Program FPGA with SoPC
2) Send the fault injection parameters to the system
3) Send the software application to the system
4) Initialize and start the fault injection
5) Execute the software application
6) Monitor the software application and compare its results to the expected values

To combine all these steps together, we developed a PC based application that can perform these steps automatically. Through our tool, we can configure the type of test we would like to do, configure the transient fault parameters, and select a software application to test. The tool will monitor one test run for a specified time with fault injection disabled, and then run a predefined number of tests with fault injection enabled. During these tests it will then compare the output of the program to the expected values. The tool will also reset the SoPC if the system does not terminate by a set time calculated from the fault free run. Once all the test runs have completed, the tool

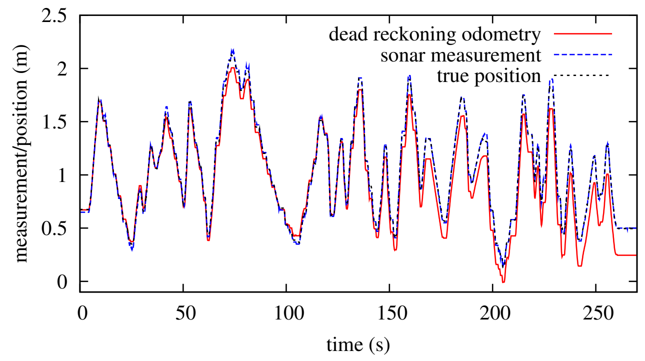

Fig. 4. Dead reckoning odometry (using wheel rotation) and raw measurement data of the robot compared to ground-truth data obtained by a laser range finder.

can output a text file containing information pertaining to each individual test, and an overall summary of all tests. With the use of Gnuplot [3], our tool can also automatically generate charts using the information gathered during each test run. This tool was used to generate all the results in Section IV.

C. Probabilistic Sensor Data Fusion

Throughout the experiments presented in this paper, we consider the problem of estimating the position x of a robot in a known environment using a Kalman filter [18]. In our application, we aim to track a mobile robot with one degree of freedom moving back and forth, e.g. in a delivery task between two stations. As a control input, the robot receives a velocity command in each time step. Furthermore, we assume the robot is equipped with a distance sensor which continually takes noisy distance measurements from its current position to one station at each time step.

The key idea of this approach is to maintain a 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]:

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

In this equation, η_t is a normalizer that ensures that $\int p(x_t \mid z_{1:t}, u_{1:t})\, dx_t = 1$. 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.

For the implementation of the described filtering scheme, we use a Kalman filter, which assumes linear system dynamics with Gaussian distributed noise. At time t the Kalman filter represents the belief $p(x_t \mid z_{1:t}, u_{1:t})$ as a Gaussian $\mathcal{N}(\mu_t, \Sigma_t)$ with mean μ_t and covariance Σ_t. The linear motion model is expressed by $x_t = A_t\, x_{t-1} + B_t\, u_t + \varepsilon_t$ and is dependent on the previous state x_{t-1}, the control input u_t and a random variable $\varepsilon_t \sim \mathcal{N}(0, R_t)$. The sensor model is also assumed to be linear, and is expressed by $z_t = C_t\, x_t + \delta_t$ where the measurement depends on the current state x_t and a random variable $\delta_t \sim \mathcal{N}(0, Q_t)$. 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 $B_t = \Delta t$, where Δt is the time elapsed since the last step of the filter. Likewise, the measurement coefficient is $C_t = 1$. The covariances R_t and Q_t of the random noise variables can be obtained in a straight forward manner by comparing recorded data to ground-truth data.

IV. Experimental Results

Our current implementation uses a Cyclone II FPGA Starter Board[2]. The entire design uses approximately a third of the FPGA (Altera EP2C20), and our SoPC runs at 12.5 MHz. The FPGA is then connected to 512 KBs of external memory which is used to store our data and test application.

Here, our tests consisted of localizing a Pioneer P3-DX [4] mobile robot while travelling back and forth between two walls. The P3-DX was equipped with a forward pointing

[2]This is an inexpensive board (100 €), and our system could easily be transfered to a larger faster FPGA, and the memory controller could be replaced with one that could handle GBytes of DDR memory.

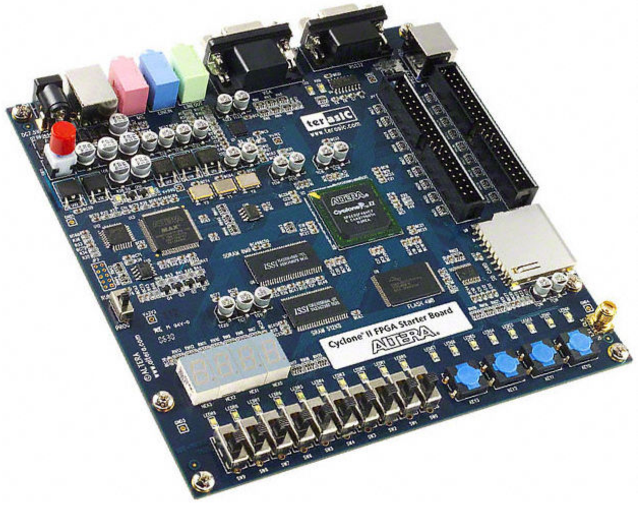

Fig. 5. Altera's Cyclone II FPGA Starter Development Kit [1].

TABLE I
RESULTS OF TRANSIENT FAULT INJECTION WHILE RUNNING THE KALMAN FILTER LOCALIZATION ON THE OURMIPS PROCESSOR.

		10 Transient Faults/s			100 Transient Faults/s		
Q-Format	Type	Time (s)	RMSE (m)	# TSF	Time (s)	RMSE (m)	# TSF
—	float	1.731	$9.83E30$	8	1.658	2.07E36	22
Q[4].[28]	fx32	0.441	0.021	3	0.441	0.022	14
Q[7].[25]	fx32	0.487	0.020	2	0.488	0.024	20
Q[10].[22]	fx32	0.558	0.019	2	0.580	0.121	21
Q[13].[19]	fx32	0.595	0.049	2	0.593	1.794	14
Q[4].[12]	fx16	0.349	0.022	1	0.343	0.022	19
Q[7].[9]	fx16	0.300	0.026	0	0.297	0.026	14
Q[10].[6]	fx16	0.257	0.036	1	0.258	0.106	17
Q[13].[3]	fx16	0.230	0.140	0	0.300	0.140	18

Devantech SRF10 miniature sonar sensor. Due to the good reflection properties of the wall the sonar sensor measurements had relatively little noise. In our experiments we recorded all the control and measurement data that the robot generated during approximately 4 minutes of operation (shown in Figure 4). We collected dead reckoning odometry (i.e. position based on the rotation of the wheels) and ground truth positions (i.e. actual physical position) with a SICK LMS 291 laser range finder. Then, we ran the Kalman filter localization algorithm (described in Section III-C) on the prerecorded data while injecting transient faults into the OurMIPS processor.

Table I summarizes the results of the Kalman filter localization algorithm using different 32 and 16 bit representations of floating point numbers. The first representation type is the normal IEEE defined 32 bit *float*. We then defined two other types of floating point numbers, mainly *fx32* and *fx16*. The fx32 format is 32 bits long, and the fx16 format is 16 bits long. The integer and decimal parts for each fx32 and fx16 type are divided as specified in the first column of Table I. For instance, Q[4].[28] means that the integer part of the fx32 variables use 4 bits, and the decimal part uses 28 bits. The fx32 and fx16 representations of course limit the range and accuracy of the floating point numbers. However, since the data generated by the sensor had a maximum range of about 2 meters with an accuracy of about one centimeter, all the representation are more than sufficient.

For each type of floating point variable type we then ran 200 transient fault simulations. In the first 100 simulations we injected an average of 10 transient faults per second of run time. For the last 100 simulations we increased the number of transient faults to 100 per second. For each group of 100 simulations we recorded the average run time in seconds per test (Time), and the root mean square error (RMSE) in meters:

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

where μ_t is the position estimate of the Kalman filter localization and x_t^* is the actual position at time t. The final piece of data that was collected in Table I is the number of times the SoPC system crashed, or became totally unresponsive due to the transient faults that were injected (# TSF).

The results in Table I first show that the time for each individual test is relatively small. In our SoPC system, since the OurMIPS processor operates at 12.5 MHz and can execute one 32-bit instruction every two clock cycles (on average), each test requires the execution of a few million instructions. Using software on a PC to simulate our entire SoPC would be considerably more time consuming. Because of this, we can rapidly perform a large number of tests.

The next important results are with respect to the calculated RMSE values. First, notice that when using normal IEEE floating point numbers, the RMSE can be huge. This is because if only one exponent bit is flipped, or falsely calculated, it can change the result by many orders of magnitudes. For fx32 and fx16 type numbers, the maximum offset that can happen is when the most significant decimal bit is flipped. So for example, the Q[4].[28] and Q[4].[12] representations are more immune to large errors triggered by transient faults than Q[13].[19] and Q[13].[3] where the decimal parts are larger. Furthermore, from this table, you can see the entire trend that the susceptibility to larger errors increases with the decimal part of the number. Changing a program to use fx32 or fx16 rather than floats is a good example of how software can be hardened against transient faults. Generally speaking, a program written to use fx32 or fx16 would typically perform significantly better than an application using IEEE floating point representations in environments experiencing many transient faults.

The last results from Table I show how many times the SoPC system failed to respond. This can be caused by a jump instruction being changed, or the PC register in the CPU being affected by a transient fault. This could cause the processor to jump into a random, uninitialized memory location where it would start to execute random instructions. Furthermore, it is also possible for bits in the control register to be flipped that results in the processor entering a forbidden state. For all of these cases in Table I, we marked them as a total system failure (# TSF) for each set of 100 runs. As can be seen, when injecting 10 transient faults per second, the system normally freezes only a few times. When increasing the number of injected transient faults, the system crashes significantly more often. 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

In this paper we presented a programmable system for testing transient faults using FPGAs. This work allows us to test the soft error susceptibility of a common 32-bit processor. Our architecture enables us to seamlessly connect our SoPC to external devices, and removes the restrictive memory limitations that previous work have had. Futhermore, our platform can execute real applications in real environments, not just random test programs. As we showed in our experimental results, our system allowed us to harden the Kalman filter localization algorithm using a better floating point representation of the data. This information would have traditionally required a very expensive radiation testing facility, and here we simulated it on a 100 € FPGA board. This application is also only a starting point. In the future, we plan to test other types of software applications while our system is directly connected to sensors with live data streams.

Lastly, since our platform can be used during the design phases of a project (before the real SoC ASIC chips are produced), and because it can run real applications, the designer can test both the hardware and software aspects of a design. This can give a designer much more flexibility when trying to meet certain FIT quality levels. For instance, under certain circumstances it might be more cost effective to meet the FIT requirements by hardening only the software as we did with our floating point representation, or more thoroughly as was presented in the software-implemented fault tolerance (SWIFT) work introduced by [25]. In other cases, error correcting hardware or the addition of redundant components to the SoC could be tested to meet more stringent soft error system requirements.

References

[1] *Altera*. http://www.altera.com/.

[2] *Altera's SignalTap II Embedded Logic Analyzer*. http://www.altera.com/literature/hb/qts/qts_qii5v3_05.pdf.

[3] *Gnuplot*. http://www.gnuplot.info/.

[4] *MobileRobots Inc*. http://www.mobilerobots.com/.

[5] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson. Generic object-oriented fault injection tool. In *International Conference on Dependable Systems and Networks*, pages 83–88, 2001.

[6] J. Arlat, Y. Crouzet, and J.-C. Laprie. Fault injection for dependability validation of fault-tolerant computing systems. In *International Symposium on Fault-Tolerant Computing*, pages 348–355, 1989.

[7] R. Baumann. Ghosts in the machine: A tutorial on single-event upsets in advanced commercial silicon technology. In *International Test Conference*, 2004.

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

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

[10] 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.

[11] 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.

[12] P. Civera, L. Macchiarulo, M. Rebaudengo, M. Sonza Reorda, and M. Violante. FPGA-based fault injection for microprocessor systems. In *Proceedings of the Asian Test Symposium*, 2001.

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

[14] 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.

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

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

[17] B. F. Dutton and C. E. Stroud. Single event upset detection and correction in Virtex-4 and Virtex-5 FPGAs. In *International Conference on Computers and Their Applications*, pages 57–62, 2009.

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

[19] H. Kobayashi, H. Usuki, K. Shiraishi, H. H. Tsuchiya, N. Kawamoto, G. Merchant, and J. Kase. Comparison between neutron-induced system-SER and accelerated-SER in SRAMs. In *International Reliability Physics Symposium*, pages 288–294, 2004.

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

[21] 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.

[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] M. Rebaudengo, L. Sterpone, M. Violante, C. Bolchini, A. Miele, and D. Sciuto. 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.

[25] 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.

[26] 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.

[27] P. Shivakumar, M. Kistler, W. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *International Conference on Dependable Systems and Networks*, pages 389–398, 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.