IEEE AUTOTESTCON 2008
Salt Lake City, UT, 8-11 September 2008

A Test Methodology for Determining Space-Readiness of Xilinx SRAM-based FPGA Designs

Heather Quinn, Paul Graham, Keith Morgan, Michael Caffrey, and Jim Krone
ISR-3 Space Data Systems, Los Alamos National Laboratory, Los Alamos, NM, 87545 USA

Abstract - *Using reconfigurable, static random-access memory (SRAM) based field-programmable gate arrays (FPGAs) for space-based computation has been an very active area of research for the past decade. Since both the circuit and the circuit's state are stored in radiation-tolerant memory, both could be altered by the harsh space radiation environment. Both the circuit and the circuit's state can be protected by triple-modular redundancy (TMR), but applying TMR to FPGA user designs is often an error-prone process. Faulty application of TMR could cause the FPGA user circuit to output incorrect data. This paper will describe a three-tiered methodology for testing FPGA user designs for space-readiness. We will describe the standard approach to testing FPGA user designs using a particle accelerator, as well as two methods using fault injection and modeling. While accelerator testing is the current "gold standard" for pre-launch testing, we believe the use of fault injection and modeling tools allows for easy, cheap and uniform access for discovering errors earlier in the design process.*

Keywords: Field programmable gate arrays, Reliability testing, Reliability estimation, Failure analysis, Space technology

I. Introduction

Field-programmable gate array (FPGA) technology, such as the Xilinx Virtex family of devices, has made inroads into space-based platforms over the past decade [1], [2]. These devices have programmable logic and routing that are used to implement user circuits and are well-suited for the digital signal processing algorithms that are often used in space. Unlike radiation-hardened anti-fuse FPGAs that can only be programmed once, radiation-tolerant devices can programmed an unlimited number of times. The ability to *reconfigure* the device to implement new circuits makes SRAM FPGAs interesting to the space community. Unlike other hardware devices that have the circuit fabricated into the silicon, new circuits can be implemented on an FPGA while on orbit. Therefore, reconfiguration can extend the usable lifetime of the system by changing the FPGA's user circuit to meet changing mission and science goals. We have also found that reconfiguration opens up many avenues for pre-launch testing of the user circuits.

Document release number: LA-UR-08-04246. This work was funded by the Department of Energy through the Deployable Adaptive Processing Systems and Sensor-Oriented Processing and Networking projects, the Cibola Flight Experiment project at Los Alamos National Laboratory, NASA through the Reconfigurable Hardware in Orbit project under LIST contract #NAG5-13516, and the Air Force Research Laboratory under the FPGA Mission Assurance Center.

Unfortunately, the SRAM technology used in these FPGAs to implement the user circuit is susceptible to radiation-induced faults, called single-event upsets (SEUs), that can affect the programmable logic and routing or affect the entire device. The best practices for FPGA-based spacecraft design encourages the use of triple-modular redundancy (TMR) in the user circuit to mask SEU-induced errors on the FPGA, in addition to error detection and correction of data stored in the device's programming memory. Not only is applying TMR an error-prone process, but sometimes the designers are unable to apply "full" TMR to the user circuit due to device size constraints. The user circuit needs to be tested pre-launch to determine whether it is working as expected, including whether TMR has been applied effectively, and whether the availability requirements are met.

The current "gold standard" for pre-launch testing of user circuits is validation through radiation experiments at a particle accelerator. For these tests the designer has the choice of using either a proton or a heavy ion accelerator, the most likely ionized particles to cause SEUs while on orbit. Fully space-qualifying a design could take days worth of time and tens of thousands of dollars at an accelerator to exercise all of the possible radiation-induced failure modes and find all of the problems with a user design. Since radiation-induced faults are statistical in nature, it may be too expensive to get good test coverage and difficult to understand how the output errors correlate to design flaws in the user design. We have found that fault injection and modeling tools are much better at providing feedback about specific design flaws to the designer. We have both a fault injection tool — the SEU Emulator — and a modeling tool — the Scalable Tool for the Analysis of Reliable Circuits (STARC) — that can be used by FPGA designers to augment radiation experiments. By using these tools, the accelerator testing is only needed for final, pre-launch validation of the user design.

In this paper we will present a three-tiered methodology that uses all of these technologies for discovering design flaws in the system before launch. The rest of the paper is organized as follows. We will explain in better detail the types of design flaws we are testing for in Section II. Section III will introduce the STARC modeling tool, which helps designers identify problems in implementing TMR in user circuits during the design stage. Section IV will cover fault injection, which helps designers do more in-depth, hardware-based testing of user designs. Section V will cover the final validation of user circuits at a particle accelerator. Given the disparate nature of these three topics, the related work for these topics will be covered in the individual sections. The paper completes with a comparison of these three methodologies in Section VI.

II. Background

SEUs affect the user circuit usually in one of two ways: by changing circuit functionality or by changing the flow of data through the circuit. SRAM bits that cause output errors when affected by an SEU are called *sensitive programming data bits* or more simply *sensitive bits*. Good testing should help designers quantify and locate sensitive bits that are the result of untriplicated logic, placement-related issues, and the application of TMR. In a fully TMR-protected circuit SEUs in the user logic should not affect the output data in the system, with the exception of some occasional placement-related issues that depend on how the circuit is implemented on the device [3]. In a design that has only had TMR partially applied to the design (*Partially TMR-Protected*), there will also be untriplicated logic and routing that will have some sensitive bits. In the remaining portion of this section will discuss how SEUs affect partially and fully TMR-protected user circuits.

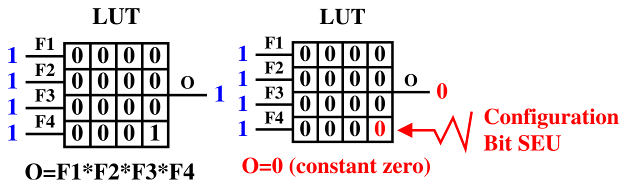
II.A Fully TMR-Protected User Circuits

In fully TMR-protected user circuits, no single-bit SEUs should cause output errors unless TMR was not applied properly or problems with logical constants exist. We have found that TMR-protected systems can be vulnerable to SEUs if the implementation of the logical constants is not carefully controlled. These logical constants are frequently used to tie off unused resource inputs, such as the “carry in” to ripple carry logic or unused address lines to a memory. With newer devices multiple bit upsets (MBUs), where a single SEU causes multiple bits to fail, have become more common [4], especially with heavy ions. We have observed MBU-induced TMR defeats [3]. These TMR defeats appear to be strongly influenced by placement issues.

II.B Partially TMR-Protected User Circuits

When TMR is only applied to a portion of a circuit due to resource constraints, SEUs can affect two different areas of the circuit: untriplicated logic and untriplicated routing. Partially TMR-protected designs could also have all of the placement-related issues that affect fully TMR-protected designs as described above.

First of all, any untriplicated logic could cause output errors to manifest when the logic is corrupted with an SEU. For example, Figure 1(a) shows a programmable logic element, called a *lookup table* (LUT), that is implementing a 4-input AND function. If the one bit that defines the “true” condition has an SEU, the result is a constant-zero function. Sometimes SEUs in untriplicated logic can be logically masked by the data the circuit is executing. For our example above, most of the possible input combinations will return the correct output. If the data in the system never exercises the one input combination that causes the error to manifest, the error will be logically masked. Output errors that manifest from untriplicated logic can only be fixed by changing the design. Therefore, the number of sensitive bits due to unprotected logic are immutable to how the user circuit is placed on the device by the design flow tools, although the location of these bits might change by rerunning the tools.



(a) Original 4-input AND Function (b) Upset LUT Function (Constant “0”)

Fig. 1. LUT Upset Example

A second set of output errors in partially TMR-protected designs stem from the programmable routing network in the untriplicated parts of the design. SEUs in the routing network changes the flow of data through the circuit. For example, an SEU in the routing network could cause an input to a LUT to float. Unlike untriplicated logic, some of the SEUs in the routing network can be influenced by the design flow tools that determine how the user circuit is placed and routed on the device. Within blocks of untriplicated logic, the design flow tools could affect the the number and location of sensitive bits in the routing network by lengthening/shortening routes or changing their location on the device.

Sometimes the logic in a design might be completely triplicated, but some or all of the input signals might not be due to the lack of I/O resources. While the input data signals will be triplicated once the data is registered the first time, the clock and reset trees will remain untriplicated. SEUs in the programmable routing along the main trunks of the clock and reset trees are very likely to affect the entire circuit, but SEUs in the leaves of these trees will often be masked if only one module in the TMR-protected design is affected. The number and location of sensitive bits can be affected by placement, but cannot be eliminated through rerunning the placement tools.

III. Modeling Tools

Reliability analysis is traditionally done with modeling tools. For designers of many types of systems, these tools allow the designers to focus on creating accurate models of their systems, instead of focusing on how to calculate the reliability. Since FPGA user circuits already have accurate functional models in terms of hardware descriptions and netlists, the right modeling tool can leverage these descriptions directly for reliability analysis, enabling the detection and correction of design flaws in the design phase where fixing flaws is significantly cheaper.

Traditionally, circuit reliability has been determined using purely analytical approaches [5] or techniques that model Boolean networks as probabilistic systems [6]–[9]. These modeling techniques represent circuits as probabilistic transfer matrices, stochastic Petri-nets, Markov chains or Bayesian networks. These analytical approaches have been found to be error-prone and computationally complex for the analysis of large designs. Similarly, a number of limitations have been identified for many modeling-based approaches. First of all, model and input data set creation greatly increase the time commitment of using these tools. Transforming circuits into intermediate probabilistic system models is an additional, computationally complex task. The complexity of calculating the circuit reliability also grows exponentially with circuit size and the number of input vector sets and the computation can take a prohibitively long time to finish. The exception to these problems is the SETRA tool [10] that directly addresses the state space issues as well as automated model generation.

For these reasons, traditional tools are not well-suited for the size of designs used in most FPGA systems. All of these limitations have led to the development of the Scalable Tool for the Analysis of Reliable Circuits (STARC) tool, which specifically addresses the limitations of model creation, input data sets and computational complexity with these solutions:

- industry-standard Electronic Design Interchange Format (EDIF) representation of a circuit as the input model,
- no input vector sets,
- memoization to reduce the computational complexity, and
- combinatorial reliability calculations.

By using the EDIF circuit representation, the designer can assess the reliability of a circuit during the design process, even if the design is not complete, the design does not work, or the hardware is not available. Without the use of input vector sets reliability is determined through the probability of device or input failure and is not dependent on specific input data sets. Without input data sets, the reliability of sub-circuits are determined by type, such as a two-bit adder, and memoized for reuse. In this manner, large-scale circuits are analyzed in a fraction of the time required by traditional approaches, making design exploration more worthwhile.

There are a few disadvantages to this approach. First, since EDIF does not contain information about the routing and the placement on the device, routing reliability is currently statistically estimated from case studies of routing placement. Furthermore, currently there is no way to assess placement-related issues, such as MBU-induced TMR defeats. We are currently working on a solution for this limitation for designs that have gone all the way through the design flow. Second, without input vector sets logic masking cannot be taken into account, and STARC estimates the worst case failure rate. While this value may be lower than the value determined by other tools [11], STARC provides a useful lower bound on the circuit's reliability.

By using the EDIF circuit representation the hierarchy in the circuit should be preserved. Since designers tend to create complex circuits by creating less complex sub-circuits, maintaining this structure can be very useful in calculating the reliability. In particular, STARC can readily exploit memoization by memoizing the reliability of sub-circuits and reusing the reliability values for sub-circuits of the same type. This reuse allows the computation to grow polynomially instead of exponentially. This hierarchical nature allows circuits to be examined at the highest level of abstraction or the most minute level of detail. STARC automatically determines the appropriate level of the hierarchy that needs to be explored. An example of this hierarchy is shown in figure 2. In this example, the reliability of components 1-4 are determine first, memoized, and then used to determine the reliability of the entire circuit.

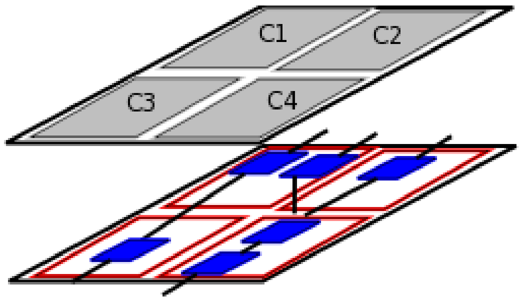


Fig. 2. Hierarchical Exploration of Circuit Design

During hierarchical exploration, dependency graphs for each primary output at each level of the hierarchy are determined. The dependency graph has all of the sub-circuits between the output and the reachable inputs. Since not all logic or inputs are reachable from every output, this technique removes unrelated logic from the reliability calculation. Once the dependency graph for an output is determined, the reliability can be calculated. In unmitigated designs, the quantity of sensitive bits is the total area of the dependency graph:

$$A(O) = \sum_{i=0}^{m} A(C_i), \tag{1}$$

where $A(X)$ is the sensitive area of X (where X is either a wire or a cell) and $C = \{C_0, ..., C_m\}$ is the set of cells that can be reached from output wire O. The reliability of basic architectural elements, such as LUTs and user flip-flops, are pre-determined and are statically loaded when STARC starts.

STARC was designed to help designers find problems in the application of TMR. For mitigated circuits, the sensitive area is confined to the part of the design that is not triplicated, as triplication will mask errors as long as there is one voter for each redundant module. There are cases where the design flow tools, in particular synthesis tools, will alter the circuit so that the TMR modules are no longer functionally equivalent or independent. In these cases two modules will share a partial calculation with the third module and the shared partial calculation becomes a single point of failure. Feedback loops in TMR-protected systems are also sensitive to *persistent errors* [12], and need to use triplication and voters to break the feedback loops. If the feedback loops are not handled in this manner, the feedback loop's state will not be able to autonomously resynchronize after the SEU is removed. In this scenario, while the first SEU in the feedback loop will be masked, another SEU in the feedback loop is not guaranteed to be masked.

In all of these cases, STARC provides warnings and information about the design to the designer. The output of the tool provides the designer a list of sub-circuits that are untriplicated, and warnings about potential single points of failures from functionally nonequivalent modules and logical constants. Since EDIF is tightly coupled to the circuit design, the designer should be able to directly use STARC's output to find and fix the design flaws in the user circuit.

IV. Fault Injection Testing

Once a design is completed and hardware is available, it is possible to move on to fault injection. Unlike modeling tools, fault injection works with the actual hardware implementation of the user circuit, allowing placement-related issues to be assessed. If designed well, a fault injection tool should have good fidelity to accelerator testing and on-orbit behavior, since the hardware and the operational behavior mimic actual usage. Finally, we would like to note that fault injection on the actual "flight" hardware is highly desirable since it is more likely to mimic or illustrate the consequences of individual upsets.

Fault injection is possible, because the interfaces that control device programming (or *configuration*) are accessible to the designer. These interfaces can be used by the designer to purposefully corrupt the programming data to mimic SEUs in programming data. While LANL designed one of the first fault injection testbeds for FPGAs with the SLAAC1-V SEU Emulator [13], since then many other organizations have created them [14]–[16]. We have also gone on to make other versions of our fault injection tool to support newer hardware devices and support MBU testing.

Fault injection tools for FPGAs all have the same basic algorithm, as shown in Figure 3. With this algorithm, faults can be injected throughout the entire programming data. It is important to run a large number of input vectors through the system after the fault is injected to avoid logical error masking. Since running a complete set of test vectors is often infeasible, our SEU emulator generates input vectors randomly so that better coverage is possible by running multiple tests on the same design. Each test provides coverage for 250,000-500,000 test vectors. It is also feasible to run a complete set of test vectors for limited portions of the circuit. Resetting and resynchronizing the user circuit after the SEU is removed is also important so that the effects of each emulated SEU is kept independent from others. Independent trials ensures that errors are properly attributed to the right programming data bit and that latent bad state from one fault injection iteration does not affect the next one.

There are usually two types of fault injection systems based on whether one or two FPGAs are used. In our SEU Emulator tool two FPGAs are used, each one hosting the same user design. Faults are injected into the *design under test* (DUT) FPGA and then run in lockstep with the same input vectors with the *golden* FPGA, which receives the same input vectors. The advantage of this system is that sharing input vectors, detecting output errors, and testing the system for resynchronization is very easy. The disadvantage is the complexity of designing the lockstep system.

In the single FPGA fault injection systems, the input vectors are run through the system twice: once without fault injection and once with fault injection. The advantage of this system is that it takes less hardware and is easier to design than a lockstep system, but the disadvantage is that the input vectors

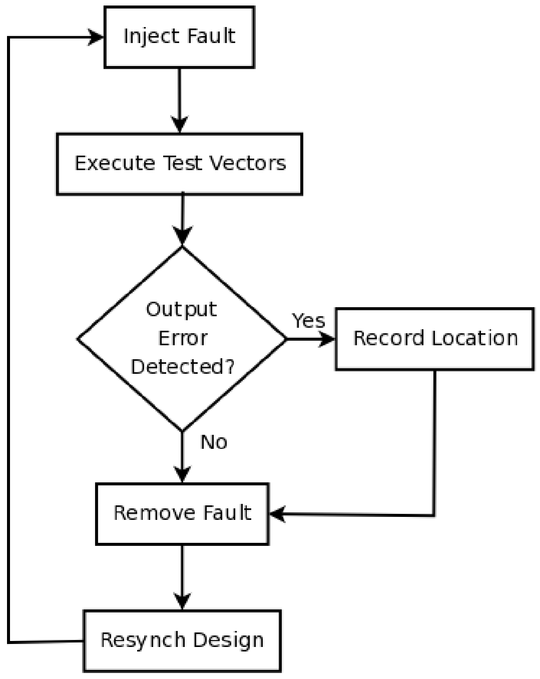


Fig. 3. Basic Fault Injection Algorithm [13]

and correct output vectors need to be saved in the system. Furthermore, determining miscompares in the output data is not simple.

In general, a good fault injection system should be able to handle different types of user circuits. With many fault injection systems, the number of clock and reset pins, the width of input and output buses, and the pin locations are often set. Due to these restrictions, sometimes the user design has to be changed to fit the fault injection system, which can reduce the usefulness of fault injection. On occasion, we have found some cases that do not lend themselves to fault injection. In these cases, the use of modeling tools is even more important.

Once fault injection is completed, the SEU locations need to be tied back to the design. If fault injection only reports a handful of errors, the designers can decide that the user circuit meets the availability requirements for the system and that further design exploration to fix design flaws is unnecessary. Unlike when using STARC, tying design problems found through fault injection to the design can be quite difficult and time consuming. While the fault injection tool returns the locations of sensitive bits, most designers do not know how to translate that location into a physical location on the device. If the physical location is determined, it is possible to use a Xilinx design flow tool, called FPGA Editor, to determine what part of the user circuit is in that location. There are times, though, that even knowing the part of the design that is causing the problem does not help. Since many errors manifest in the routing network of TMR-protected designs, it is possible the fault is caused by a signal that is passing through a routing switch, rather than an error in the user logic.

V. Accelerator Testing

One of the advantages of doing accelerator testing after the use of fault injection and modeling tools is that the designers should be better prepared for accelerator testing. To this end, the designers should know the areas of the circuit design that should cause output errors from the modeling tools and know the locations of these faulty areas through fault injection. Furthermore, if the designer has been using a lockstep fault injection tool, the fault injection hardware can be used as the test fixture for the accelerator tests. Even the lockstep fault injection software can be used as part of the accelerator test fixture with minor modifications. If a lockstep system was not used for fault injection, a test fixture that can easily detect output errors is highly desirable so that real-time feedback is available during the test to ensure that the test is functioning properly. A number of FPGAs will be needed for the test, since the parts are only guaranteed to operate properly up to 100 KRads of ionizing dose.

The algorithm for the software aspect of the test fixture is very similar to the fault injection tool's algorithm. Instead of injecting faults artificially, though, the particle accelerator will be injecting radiation-induced SEUs. Unlike fault injection, controlling the number of upsets that occur during one loop of the algorithm is more difficult. SEU removal and SEU-induced single-event functional interrupts (SEFIs) that affect the functionality of the entire device complicate testing. These three problems will be discussed below.

The arrival time of radiation-induced faults are a Poisson random processes. As the designer will want to reduce the probability of multiple independent upsets (MIUs) causing an output error, the beam's flux is tuned so that on average only one SEU occurs per algorithm loop. Even still, Poisson statistics tell us that, even if the beam's flux is tuned to one SEU per algorithm loop, there is a 37% chance that no SEUs occur, a 37% chance that one SEU occurs, and a 26% chance that two or more SEUs occur during the given time period. Since not all SEUs cause output errors and only a few sensitive bits might exist, it can take some time for errors to manifest. The formula for determining the time interval for one output error to manifest is:

$$\begin{aligned} N_{OE} &= \left(\frac{N_{bits}}{N_{device}}\right)^{-1} && (2) \\ T_{OE} &= N_{OE} \times T_s, && (3) \end{aligned}$$

where N_{OE} is the number of samples until an output error, N_{bits} is the number of sensitive bits, N_{device} is the total number of bits in the device, T_s is the time length for each sample, and T_{OE} is the average time span until an output error.

Removing SEUs quickly is important so that the user circuit can recover before the next SEU occurs. There are two ways to remove an SEU during an accelerator test. First, the SRAM FPGA can be completely reprogrammed through *off-line* programming, where the FPGA is taken off line for the express purpose of reconfiguring the device. Taking the device off line tends to be costly in terms of time, but often needed in the case of SEFIs, where a full reprogramming of the device is the only reliable method for restoring the FPGA to a known, functional state. A second approach available to Xilinx SRAM FPGAs is to use *on-line* programming capabilities. In this case, the FPGA remains operational while its programming data is repaired. Using on-line reprogramming, it is possible to either completely rewrite all of the programming data or to only fix the portions that have SEUs. This later case is safer since it affects the least amount of programming data at a time and since the FPGA's programming circuitry can be affected by SEUs. To reduce the time required to identify an SEU and fix it, we recommend the use of external SEU detection hardware as opposed to software.

After the accelerator test is completed, the results need to be examined so that correlations between output errors and known sensitive bits can be determined. Since the SEUs in accelerator testing do not present themselves in the system uniformly or at specified time intervals, correlating output errors to specific SEU locations can be a challenge. In some cases, the output error follows the SEU by several algorithm loops and other times the reporting software will output the existence of the output error before the SEU location. On other occasions, some output errors need to be dropped from the data set, such as when the incidence of multiple independent upsets are the cause of the output error. Often times all of the results around a SEFI event will need to be ignored, since removing the SEFI is time consuming and the system will likely report output errors for several iterations until the circuit state resynchronizes.

As long as the user circuit that is being tested is the same one tested in fault injection, the results from fault injection can be used to disambiguate the accelerator test results. Due to the problems described with attributing SEUs to output errors, the most effective approach for analyzing accelerator results is to look at several SEU locations before and after the output error in the log. This "window" of SEU locations can then be compared to fault injection results to determine if any of these SEU locations caused an output error in fault injection. While this method can usually help a designer correlate output errors with fault injection results, some output errors cannot be completely correlated. In some cases, the errors are due to SEUs in user memory, such as flip-flops, and not due to programming data upsets. The number of these types of errors can be predicted based on the amount of user memory used in the design and the susceptibility of these memory elements to SEUs. In other cases, sometimes the accumulation of errors in the circuit state caused the output error. For these cases, sometimes part of an accelerator test can be "played back" using the fault injection tool, where the tool uses the accelerator log to inject faults in specific locations in a particular order. Through this attribution process, the designer can determine whether the output error can be explained and whether further design exploration is needed to address potential design flaws.

For fully or mostly mitigated designs, accelerator testing should be uneventful and the user circuit should be able to operate for minutes or longer without any output errors. For example, using Equation 3, if fault injection only found 100 sensitive bits in a device with 75 million bits, at one algorithm loop per second the first output error is only guaranteed to occur randomly within a 208 hour time span. Some designers will do multiple rounds of tests with different flux levels and different durations. In particular, one test might be very low flux over several hours, mimicking average operation on orbit, and another test might have a very high flux over a couple of minutes, mimicking solar flare conditions or to otherwise reduce test time. If, at the end of these tests, the design is able to operate either error-free or within the availability requirements, the design is considered space ready.

If the error rate is much higher than indicated by the fault injection tool, either the flux could be too high or there might be problems with either the fault injection or accelerator test fixture. When designing new fault injection and accelerator test fixtures it is important to test the setup by correlating output errors to the source of the errors to ensure fault injection works, as well as correlating the results of fault injection and the accelerator tests. If the results cannot be correlated, then the methodologies for both systems need to be examined.

VI. Results

In this section, we will compare the use of these three methodologies on a circuit. The circuit, an adder tree, is fully triplicated and was designed originally to test for placement-related issues due to both MBUs and logical constants. This design was implemented for a Xilinx Virtex-II FPGA (XC2V1000). All three methodologies were used on this design. In the following paragraphs, we will describe the amount of time, the quality of the results, and the cost of using these methodologies.

In terms of time, STARC is comparatively much faster than the other two methods. Within a minute, the tool returned the result that the design was triplicated properly and with warnings that placement-related issues could exist from logical constants. As STARC cannot currently estimate the placement-related issues, it is unable to estimate how many bits in the design could cause output errors. In terms of cost, STARC is free to government users.

In terms of test coverage, the SEU Emulator was much more complete than the other two methods. With fault injection, we were able to find 285 single-bit SEU locations, 18,733 2-bit SEU locations, 11,264 3-bit SEU locations, and 19,464 4-bit SEU locations that cause the design to output bad data. Each pass through the fault injection test takes two hours per run and each MBU test is a separate test. As the MBU tests are run with specific MBU shapes based on our analysis of how MBUs affect the Virtex-II, we were able to constrain the MBU tests to the six most common shapes. In all, fault injection tests took 14 hours for the seven tests. In terms of cost, the fault injection hardware is about $6,000 and the software is free to government users.

As validation for both of these tests, we did a two hour long test at at the University of Indiana Cyclotron Facility's proton accelerator. During this test we were able to observe 50 output errors, of which 21 were attributed to SEFIs, 13 were attributed to MIUs, and 16 were attributed to the two phenomenas that we were looking for. Of the 16 output errors, 88% we were able to later correlate to known fault injection error locations. At three algorithm iterations a second, we would have been able to test all of the single bit errors in no less than 16 days, assuming that no single-bit fault location was exercised multiple times. As is, we were able to test at a higher flux to be able to shorten the time frame of the test considerably. Since the MBU-related issues have only a 2% chance of occuring due to proton radiation, completing the 2-bit test would have taken over two years. In terms of cost, we were able to use the hardware and software from fault injection and only had to pay the accelerator fees of $1,200 for two hours of test time and $500 for the FPGA. Had we completed the single-bit test, we would have to pay for at least 385 hours of testing and 192 FPGAs for a total cost of $288,000.

Since we shortened our accelerator test considerably, the initial cost of the hardware for the fault injection tool is the highest of the three test methodologies. Had we done a complete validation of the user circuit, though, the accelerator test would have been the most expensive. Also, it should be noted that the cost of the fault injection tool is amortized across all of the fault injection tests and the accelerator testing. Since the hardware infrastructure can be reused an unlimited number of times, if the FPGA is not irradiated, the cost is reasonable. When the test coverage is factored in, the amount of time and cost invested in the fault injection tool is the best option. While fault injection should never replace accelerator testing, the accelerator test was shortened when we were able to confirm our fault injection results. We also believe that using STARC decreased the overall time commitment and iteration that takes place in fault injection and accelerator testing. Finally, since the design had TMR properly applied to the circuit from the beginning, which was confirmed throughout the testing, there was no need to determine what was wrong with our design.

VII. Conclusions

In this paper we presented a three-tiered methodology that finds design flaws in FPGA user circuits and locates the source of the faults on the FPGA. One methodology used the circuit representation to find design flaws through modeling. The second methodology used fault injection to locate how the design flaws translated to physical locations on the FPGA. The final method was an accelerator test to validate the previous results. We also showed how these three methodologies compared in terms of test coverage, time, and cost. While the modeling tool was the fastest, fault injection was the best methodology in terms of cost and test coverage.

References

[1] E. Fuller, M. Caffrey, P. Blain, C. Carmichael, N. Khalsa, and A. Salazar, "Radiation test results of the Virtex FPGA and ZBT SRAM for space based reconfigurable computing," in *Proceeding of the Military and Aerospace Programmable Logic Devices International Conference(MAPLD)*, Laurel, MD, September 1999.

[2] M. Caffrey, M. Echave, C. Fite, T. Nelson, A. Salazar, and S. Storms, "A space-based reconfigurable radio," in *Proceedings of the 5th Annual International Conference on Military and Aerospace Programmable Logic Devices (MAPLD)*, September 2002, p. A2.

[3] H. Quinn, K. Morgan, P. Graham, J. Krone, M. Caffrey, and K. Lundgreen, "Domain crossing errors: Limitations on single device triple-modular redundancy circuits in Xilinx FPGAs," *IEEE Transactions on Nuclear Science*, vol. 54, no. 6, pp. 2037 – 43, 2007.

[4] H. Quinn, P. Graham, J. Krone, M. Caffrey, and S. Rezgui, "Radiation-induced multi-bit upsets in SRAM-based FPGAs," *IEEE Transactions on Nuclear Science*, vol. 52, no. 6, pp. 2455 – 2461, December 2005.

[5] J. A. Abraham and D. P. Siewiorek, “An algorithm for the accurate reliability evaluation of triple modular redundancy networks,” *IEEE Transactions on Computers*, vol. 23, no. 7, pp. 682–692, July 1974.

[6] S. Krishnaswamy, G. F. Viamontes, I. L. Markov, and J. P. Hayes, “Accurate reliability evaluation and enhancement via probabilistic transfer matrices,” in *Design, Automation and Test in Europe (DATE’05)*, vol. 1. New York, NY, USA: ACM Press, 2005, pp. 282–287.

[7] C. Hirel, R. Sahner, X. Zang, and K. Trivedi, “Reliability and performability using SHARPE 2000,” in 11^{th} *Int’l Conf. on Computer Performance Evaluation: Modeling Techniques and Tools*, vol. 1786, 2000, pp. 345–349.

[8] G. Norman, D. Parker, M. Kwiatkowska, and S. Shukla, “Evaluating the reliability of NAND multiplexing with PRISM,” *IEEE Transactions on CAD*, vol. 24, no. 10, pp. 1629–1637, 2005.

[9] F. V. Jensen, *Bayesian Networks and Decision Graphs*. New York: Springer-Verlag, 2001.

[10] D. Bhaduri, S. K. Shukla, P. S. Graham, and M. B. Gokhale, “Reliability analysis of large circuits using scalable techniques and tools,” *IEEE Transactions on Circuits and Systems - I: Fundamental Theory and Applications*, vol. 54, no. 11, pp. 2447 – 60, November 2007.

[11] D. Bhaduri and S. Shukla, “NANOLAB—a tool for evaluating reliability of defect-tolerant nanoarchitectures,” *IEEE Transactions on Nanotechnology*, vol. 4, no. 4, pp. 381–394, 2005.

[12] K. Morgan, M. Caffrey, P. Graham, E. Johnson, B. Pratt, and M. Wirthlin, “SEU-induced persistent error propagation in FPGAs,” *IEEE Transactions on Nuclear Science*, vol. 52, no. 6, pp. 2438 – 45, 2005.

[13] E. Johnson, M. Caffrey, P. Graham, N. Rollins, and M. Wirthlin, “Accelerator validation of an FPGA SEU simulator,” *IEEE Transactions on Nuclear Science*, vol. 50, no. 6, pp. 2147–2157, December 2003.

[14] M. Alderighi, F. Casini, S. D’Angelo, M. Mancini, S. Pastore, G. Sechi, and R. Weigand, “Evaluation of single event upset mitigation schemes for sram based fpgas using the flipper fault injection platform,” in *Proc. of the 22th IEEE Int. Symp. Defect and Fault Tolerance in VLSI Systems (DFT07)*, September 2007, pp. 105–113.

[15] M. Berg, C. Perez, and H. Kim, “Investigating mitigated and non-mitigated multiple clock domain circuitry in a Xilinx Virtex-4 field programmable gate arrays,” in *Single-event effects symposium*, 2008.

[16] G. Swift, C. W. Tseng, G. Miller, G. Allen, and H. Quinn, “The use of fault injection to simulate upsets in reconfigurable FPGAs,” 2008, submitted to the military and aerespace programmable logic devices conference (MAPLD08).