Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2012, Article ID 850487, 11 pages
doi:10.1155/2012/850487

*Research Article*

# A Fault Injection Analysis of Linux Operating on an FPGA-Embedded Platform

**Joshua S. Monson, Mike Wirthlin, and Brad Hutchings**

*Department of Electrical and Computer Engineering, Brigham Young University, 459 Clyde Building, Provo, UT 84602, USA*

Correspondence should be addressed to Joshua S. Monson, jsmonson@gmail.com

Received 1 May 2011; Revised 28 July 2011; Accepted 1 September 2011

Academic Editor: Claudia Feregrino



An FPGA-based Linux test-bed was constructed for the purpose of measuring its sensitivity to single-event upsets. The test-bed consists of two ML410 Xilinx development boards connected using a 124-pin custom connector board. The Design Under Test (DUT) consists of the "hard core" PowerPC, running the Linux OS and several peripherals implemented in "soft" (programmable) logic. Faults were injected via the Internal Configuration Access Port (ICAP). The experiments performed here demonstrate that the Linux-based system was sensitive to 199,584 or about 1.4 percent of all tested bits. Each sensitive bit in the bit-stream is mapped to the resource and user-module to which it configures. A density metric for comparing the reliability of modules within the system is presented. Using this density metric, we found that the most sensitive user module in the design was the PowerPC's direct connections to the DDR2 memory controller.

## 1. Introduction

Over the last decade, Linux Operating Systems (OSs) have been used on several space-based computing platforms. NASA, for example, sponsored the FlightLinux project which culminated by demonstrating a reliable Linux OS on the UoSat12 satellite [1]. The use of Linux on the UoSat12 provided enough compatibility with ground systems to allow the satellite to be accessible over the Internet. Compatibility with ground systems is only one of the many reasons to use Linux on space-based computing platforms.

Reliable hardware is essential for Linux to operate; however, integrated circuits (ICs) aboard space-based computing platforms are susceptible to failures known as Single-Event Upsets (SEUs). SEUs are random bit flips caused by high-energy particles that collide with the ICs. To ensure correct operation in the presence of radiation, ICs can be specially designed or "hardened." Unfortunately, radiation-hardened ICs are expensive and usually two or three silicon generations behind the state of the art [2]. These factors limit the use of radiation-hardened parts in space-based computing and leave engineers looking for alternatives.

Field Programmable Gate Arrays (FPGAs) are among the state-of-the-art components that are of interest in space-based computing. FPGAs are microchips that contain an array of logic and interconnect that can be programmed and reprogrammed to perform almost any digital function. FPGAs often replace application-specific integrated circuits (ASICs) in space-based computing because designing for FPGAs is faster and less expensive than designing for ASICs. Additionally, reprogrammability allows designers to remotely fix bugs that appear after the platform has launched. These features make FPGAs ideal for space-based platforms [3–5].

The function performed by reprogrammable FPGAs is defined by the values of memory cells on the device known as the bit-stream. Changing the values of the bits in the bit stream may modify the behavior of a logic circuit on an FPGA (until it is reconfigured). For example, a single change in the bit stream has the potential to change the contents of a Look-Up Table (LUT), connect nets together, or completely disconnect a net. In most ICs, SEUs will only corrupt data. In FPGAs, however, SEUs are able to affect data and the logic function performed.

To prevent SEUs from affecting the output of an FPGA design, mitigation techniques are used. Common methods of mitigating and detecting the effects of SEUs are bit-stream scrubbing [6], Triple Modular Redundancy [7] (TMR), Partial TMR (PTMR) [8], and duplicate with compare (DWC) [9]. TMR uses redundant circuits and majority voters to improve the reliability of FPGA designs. PTMR is a reduced form of TMR that has been shown (on small circuits, at least) to reduce area overhead with only a small reduction in reliability. In bit-stream scrubbing, the bit-stream of the FPGA is occasionally rewritten to prevent the accumulation of SEUs. DWC is a method of detecting SEUs by creating a copy of the circuit, comparing the results, and flagging differences when they occur.

In FPGA-Embedded Linux systems using "hard core" processors, important peripherals (such as the memory controller) are implemented in the reconfigurable fabric of the FPGA and are susceptible to SEUs. SEU-induced failures in these components have the potential of crashing the kernel. Understanding each peripheral's likelihood of causing a kernel failure due to an SEU aids in understanding the reliability of the system and in creating a more reliable system at the lowest cost. To gain this understanding, we constructed a test-bed that allows us to simulate SEUs in the FPGA fabric surrounding a "hard core" embedded processor running a Linux kernel using a process known as fault injection [10]. Preliminary results, along with a fault injection analysis, from this test-bed have been previously reported [11]. In this paper, we present new results from a fault injection test using the same design under test. For this new test, we have improved both our fault injection process and analysis. These changes have resulted in a 3X improvement in detected sensitive bits and a decrease (from 8% to less than 1%) in sensitive bits that could not be mapped to part of the design.

## 2. Previous Work

The development of fast, comprehensive fault injection systems has been the focus of much of the previous work [12–14]. These fault injection systems are able to emulate upsets in FPGA fabric and microprocessor cache lines and special registers. Rather than performing exhaustive tests, these fault injection systems use a probabilistic model to statistically determine the reliability of a design.

The work presented by Sterpone and Violante [15] performed fault injection experiments on the memory image of a Linux microkernel running on a Xilinx Microblaze processor implemented in soft-logic. Their work focused on memory faults during the bootstrapping process but did not examine the sensitivity of the circuitry/logic that implements their system. Essentially, their fault-injection process modified the content of the memories that contain the Linux program. In contrast, our effort injects faults directly into the hardware implementation and analyzes the sensitivity of a full Linux kernel system (rather than a micro kernel) to circuit and logic failures that may be caused by SEUs.

In another work by Johnson et al. [16], they describe the validation of a fault injection simulator using a proton accelerator. Their analysis used the bit-stream offsets of sensitive bits to compute the row and column locations of the sensitive bits. Doing this allowed them to create a "map" of the FPGA showing the locations of sensitive bits. Their fault injection simulator was able to predict the locations of an impressive 97% of the upsets caused by the proton accelerator.

In this work, we also endeavor to improve the verification of fault injection experiments. For our fault injection experiment, we studied the bit-stream to determine the relationships between configuration bits and FPGA resources; this allowed us to take the next logical step and map sensitive bits to FPGA resources and, further, map resources to user-design modules.

Sterpone and Violante also used knowledge of the configuration bit-stream, but rather than examining what happened (as we do) they tried to predict where faults would occur. In [17, 18], they present a reliability analysis tool that would perform a static analysis to predict locations of sensitive bits then they would perform fault injection based on their predictions. They found that their static analyzer could predict the locations of all the sensitive configuration bits in a design mitigated using TMR without being overly pessimistic. They also found that their partial fault injection results matched that of an exhaustive fault injection test.

The similarity between our work and their work is the reliance on architectural and configuration bit-stream knowledge to identify where the problems might occur. A key difference is that their work focused on verifying relatively small systems mitigated by TMR, while our work focuses on the reliability analysis of large, unmitigated FPGA designs. It is likely that their techniques could be used to analyze this Linux system but unfortunately, their system is not available to us and direct comparisons between the two approaches cannot be made.

## 3. System Architecture

The process of fault injection requires a development board with the ability to feed test vectors into a design under test, monitor the important outputs, and modify the configuration memory of the design. The ideal development board to perform these functions would contain three FPGAs, one to act as the golden, one to act as the device under test, and one to act as the experiment controller. Should the output of the golden be known or simple, a two-FPGA solution would be sufficient. These FPGAs should be connected such that the experiment controller is able to access the configuration memory of the device under test and send (receive) input (output) vectors from both the golden and design under test. Unfortunately, there are very few boards that provide the required connectivity. This is the primary reason we have developed our own fault injection platform.

Our two-FPGA fault injection system consists of a Linux-based host PC, two Xilinx ML410 development boards, and a 124-pin custom connector board. In our system, one ML410 acts as the experiment controller and golden and the other acts as the Device Under Test (DUT). The connector board provides enough connectivity for the controller to

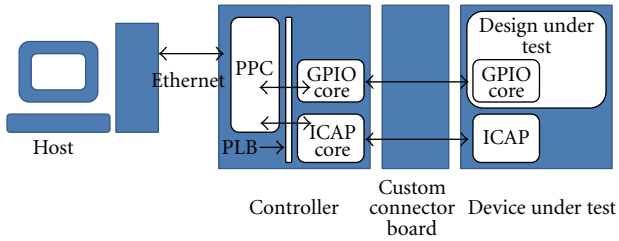


Figure 1: A block diagram of our fault injection system.

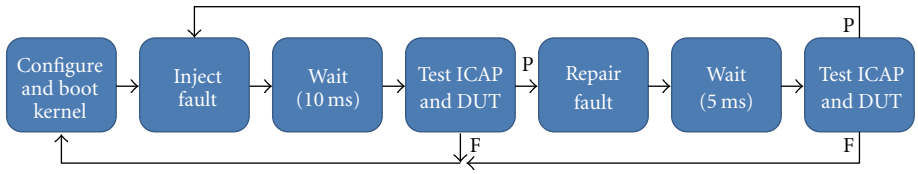


Figure 2: Flowchart of the fault injection routine.

send (receive) input (output) vectors from the DUT and also provides enough pins to fully connect the experiment controller to the Internal Configuration Access Port (ICAP) of the DUT. The ICAP is a module that resides inside the FPGA fabric and provides connections that allow components inside the FPGA to read and write the configuration memory.

As shown in Figure 1, the Linux-based host provides a simplified interface to the fault injection platform over an Ethernet connection. This allows the host to maintain an NFS mounted file system that provides a simple means for transferring software, test vectors, and results to and from the experiment controller. This connection allows experiment software to be developed on the host and then cross-compiled and transferred to the PowerPC 405. The RS-232 serial port acts as the console for the Linux kernel.

All DUT test designs communicate directly with the experiment controller via routing (contained in the DUT test design) that connects the DUT's ICAP to the custom connector board. The experiment controller injects faults through this 400 MB/sec connection. In cases where an injected fault causes the DUT's ICAP interface to fail, the controller can completely reconfigure the DUT using the external configuration interface. The control and communication circuitry for the ICAP is implemented on the experiment controller board to maximize available circuitry in the DUT. The ICAP interface on the experimental-controller board is controlled and configured by software and supports a wide variety of experiments.

## 4. Description of Fault Injection Experiment

The goal of this fault injection experiment is to measure the sensitivity of an FPGA-based Linux system to upsets in the configuration memory. In this FPGA-embedded system, all inputs and outputs of the processor must travel through the FPGA fabric. For example, the clock, memory interface, STDIN and STDOUT, general purpose I/O, and reset circuitry are implemented in the FPGA fabric. Thus the processor is unable to operate independently of the FPGA fabric, and faults injected into these components may cause the processor or Linux kernel to fail.

In general, fault injection tests are performed by modifying the configuration memory of the FPGA and then testing or observing the design to determine if the modification has caused a failure. The configuration memory is modified by reading a frame (the smallest addressable portion of configuration memory), changing a bit within the frame, and writing the frame back into the device. If the modified configuration memory causes the DUT to fail, the bit is considered "sensitive", otherwise the bit is considered "not sensitive."

In this test, each configuration bit is upset for over $10^6$ FPGA clock cycles. However, it is not possible to test each bit in all possible states of the processor, I/O, and operating system. It is possible that some configuration bits tested during fault injection may be tagged as nonsensitive but are in fact sensitive under certain system conditions. While we recognize that some sensitive configuration bits may be tagged as nonsensitive, this form of fault injection provides a good estimate of the "average" sensitivity behavior of the system under test. Previous results from similar fault injection experiments show that the vast majority of sensitive bits are easily detected (i.e., they have a high probability of detection) and the average sensitivity of a design is adequately represented in fault injection experiments [16].

This fault injection procedure is carried out by the controller as illustrated in Figure 2. Before starting the first test, the controller reconfigures the DUT and waits for the Linux kernel to boot and the test program to start. After the DUT has booted, the controller injects a single fault into the configuration memory of the DUT. After 10 ms, the DUT is tested five times over an interval of 1 ms before the fault is classified. If the DUT fails to respond to any of the tests, the bit is classified as sensitive and the DUT is reconfigured and rebooted. On the other hand, if the DUT successfully responds to all five tests, the bit is classified as not sensitive.

Table 1: Time breakout of a nonsensitive bit test.

| | |
|---|---|
| Configure FPGA | 4 sec |
| Boot Linux Kernel | 16 sec |
| Inject fault | .226 ms |
| 10 ms wait | 12.5 ms |
| Test ICAP | .105 ms |
| Test DUT | 17.9 ms |
| Repair fault | .120 ms |
| 5 ms wait | 8.9 ms |
| Test ICAP | .104 ms |
| Test DUT | 17.9 ms |
| Avg. test time (non-sensitive bit) | 59.7 ms |

To prevent the accumulation of faults, the hardware and software must be returned to a correct state before beginning the next test. This can be done by performing a full FPGA reconfiguration, recopying the original Linux image into memory and rebooting the kernel. While this is the ideal approach, the process of reconfiguring, copying the Linux memory image, and booting the kernel takes 20 seconds. Since 98% of the more than 13 million configuration bits tested are not sensitive, rebooting the DUT after each bit would cause the test to be prohibitively long. The alternative to rebooting the DUT is to repair the fault and move onto the next test. This approach does not guarantee that the system state is error free and in fact it may be possible for the repair of a fault to induce an error. To ensure that the system is at least working correctly, we subject the DUT and ICAP to another set of tests after a 5 ms wait period. If a fault is found, it is attributed to the bit most recently tested and the entire DUT is rebooted.

Sensitive configuration bits are detected in the DUT using a simple two-phase hand shaking protocol over the general purpose I/O connections (GPIO) provided by the custom connector board. The controller initiates a test of the DUT by inverting its GPIO bit. The DUT responds successfully to the test by inverting its GPIO bit to match the controller's GPIO bit. The DUT implements this protocol with a small test program that runs in the background of the Linux kernel. The pseudo-code for this test program is shown in Listing 1. The test program accesses the GPIO module to see if the controller inverted its bit; if the controller has inverted its bit, the test program responds by inverting the DUT's GPIO bit to match. At the end of each iteration, the test program is supposed to sleep for 200 $\mu$s and wake up and repeat the process; however, time measurements of the test program reveal that the program really sleeps for close to 4 ms. Using the top program, we found that the Linux kernel spends 99% of its time in the idle state and 1% of its time for user and system processes.

Table 1 shows the average time for each phase of the test of a single non-sensitive bit. The total testing time of a non-sensitive bit is almost 60 ms with the majority of the time occupied receiving 5 test responses from the DUT. Note that the wait times are slightly longer than initially intended because of the use of the `usleep()` function.

```
dut_test_program ( )
{
     int   dut_gpio = 0;
     int   controller_gpio;
     init_gpio ( );
     while ( 1 )
     {
          controller_gpio = read_gpio ( );
          if( gpio_value != dut_gpio)
          {
             dut_gpio = controller_gpio;
             set_gpio (dut_gpio);
          }
          usleep (200);
     }
     close_gpio ( );
}
```

Listing 1: Pseudo-code of DUT test program.

To convince ourselves that the 5 and 10 ms wait times imposed after faults were injected were sufficient to detect the vast majority faults, we performed additional tests with longer wait times. A subset of the configuration memory (4 different configuration blocks) was tested with an additional 15 ms (2X) and 30 ms (3X) of wait time. Table 2 summarizes the number of sensitive configuration bits found in each test. As Table 2 shows, increasing the length of the test did not significantly increase the number of sensitive bits found. Additionally, we found 97.8% percent of the sensitive bits were identical for all three tests which suggests that the 10 ms wait time is adequate.

Although fault injection is a proven and effective way to emulate SEUs, there are a few FPGA components that fault injection cannot test. For example, reading and writing the block rams (BRAMs) via the ICAP may cause side effects that artificially increase the bit-sensitivity count. Some LUTs in the Virtex 4 architecture contain additional circuitry that allows them to be used as 16-bit shift registers (SRL16s) or distributed RAM memories (LUT RAMS). The dynamic memory elements of SRL16s and LUT RAMs are part of the configuration memory and can be corrupted during reads and writes. To prevent this, Xilinx has included the GLUTMASK as a configuration option in the Virtex 4 architecture. When set, the GLUTMASK prevents both configuration memory reads and writes from modifying the contents of SRL16s and LUT RAMs. IOB (input/output buffer) configuration blocks should also be avoided; these bits control FPGA I/0 pins and arbitrarily flipping these bits may cause board or device damage. All of these situations were taken into account during the design of our experiment.

## 5. Sensitive Bit Density Metric

When designing for FPGAs, the goal is often to squeeze as much performance out of the device as possible. Reliability

Table 2: Number of sensitive configuration bits detected in original and 2X and 3X wait time tests.

| Block | Original test | 2X wait times | 3X wait times |
|---|---|---|---|
| 1 | 495 | 494 | 494 |
| 2 | 478 | 456 | 462 |
| 3 | 494 | 494 | 494 |
| 4 | 577 | 579 | 569 |

requirements often make this goal difficult to achieve because of the area overheads of mitigation techniques such as TMR.

A way to reduce the area cost of reliability is to use lesser mitigation techniques such as PTMR or DWC. While the area costs of these methods are lower they may have other costs such as lower performance or reliability. In some systems, costs may also be reduced by applying different mitigation techniques to different modules in the design. How should an engineer decide which mitigation techniques should be applied to each module to make the most efficient use of the FPGA?

One method would be to perform a fault injection test and determine which modules contain the most sensitive bits and apply more aggressive mitigation techniques to the modules that contain the most sensitive bits.

While this method is simple and straightforward it may not lead to the most efficient use of area. For example, consider a fault injection test of a design containing a large module that is moderately resilient to injected faults and a small module that is unable to tolerate a single fault at any time. After the test, it may be found that the large module, because of its size, contained more sensitive configuration bits than the small module. However, because the large module masks a portion of its sensitive bits while the small module does not, redundancy techniques would be more efficiently applied to the small module even though it has fewer sensitive bits. By using more redundancy techniques on the smaller module, more sensitive bits will be mitigated per unit area than using the same mitigation technique on the larger module.

*5.1. A New Metric.* To deal with this issue, we present a metric that takes into account both the number of sensitive bits and size of the module. This metric is called the sensitive bit density metric. This metric was first introduced by us in [11] and was measured in sensitive bits per unmitigated resource. The resources considered were nets and instances. Unfortunately, nets can vary in length while instances can vary in configuration, thus using nets and instances obscures the size of the sensitive cross-section of configuration bits. In this paper, we have improved the accuracy of the sensitive bit density metric by measuring it in sensitive bits per configuration bit. This directly links the metric to the sensitive cross section of the design. Modules with a higher sensitive bit density metric will be more efficiently mitigated by redundancy techniques than modules with lower sensitivity bit metrics.

To calculate the sensitive bit density metric, we must know the number of sensitive configuration bits and the bit-area of each module in the design. Fault injection is used to estimate the number of sensitive configuration bits of each module in the design. Ideally, the bit-area of the each module is calculated by adding up the configuration bits that must have a specific value for the design to work properly. To make our estimate of design bit-area for modules in our Linux system, we have added up all the bits in used routing resources and the bits required to set components of slices that are explicitly used.

One might ask, why do not all of the bits actually demonstrate sensitivity during a test? It is because sensitivity is a dynamic phenomenon that depends, to some extent, on system behavior. In our Linux system, there are three issues that reduce the number of detections of sensitive bits: masking, hiding, and kernel resiliency.

*5.2. Masking, Hiding, and Kernel Resiliency.* Masking occurs when the current operating mode of the circuit does not allow the invalid signal to propagate and cause a system failure. A well-known form of masking is TMR. In TMR, majority voters mask the effects sensitive bits that reside in one of the three redundant copies of the circuit. In our system, many modules are addressed over the processor local bus (PLB). If these components are not used by the processor, they cannot cause a failure in the Linux kernel and are thus masked from causing a failure.

Hiding occurs when the SEU causes the value of the affected signal to be correct during the fault injection test. For example, consider a circuit with a low asserted reset signal and assume that the reset is asserted infrequently. Suppose an SEU strikes a portion of the routing causing an open which results in a "stuck at" 1 fault. This circuit will continue to operate correctly until a reset is required. During the time the circuit is operating correctly, we would say that this sensitive bit was hidden.

The kernel is said to be resilient to faults that are propagated into the processor but that do not cause a kernel failure. In our opinion this is most likely to occur when a fault appears in a data or instruction word before it is consumed by the processor. A data word may not effect the operation of the kernel at all, while a fault could appear in an unused field of an instruction word.

## 6. Analysis of Results

Our results will be presented using three different analyses. The general overview analyzes the overall number of sensitive bits and describes the reliability of our unmitigated Linux System. In the FPGA resource analysis, the sensitive bits analyzed in the general overview are mapped to the resources they control. In the user circuit analysis, the sensitive resources are mapped to the modules that contain them (memory controller, UART, etc.) and a metric for comparing the sensitivity of modules is explained.

These three analyses are interesting because they provide a designer with data that answers questions such as what is

Table 3: Device logic utilization.

| Resource | Used | Available | Utilization |
|---|---|---|---|
| Slice registers | 6,271 | 50,560 | 12% |
| 4-input LUT | 5,688 | 50,560 | 11% |
| Block RAMs | 53 | 232 | 23% |
| SRL16 shift registers | 281 | 50,560 | .5% |
| LUT RAMs | 0 | 50,560 | 0% |

the general sensitivity of the DUT? What resources contain the most sensitive configuration bits? Which modules in the DUT are most sensitive to SEUs? Answers to these questions could aid in the directed application of mitigation techniques and possibly save design area or suggest FPGA architecture improvements.

In addition, these analyses allow those performing fault inject experiments to verify their results by confirming that sensitive bits actually have the ability to effect the design. This is done by mapping sensitive bits to actual resources in the FPGA and verifying that they belong to or can affect a component in the DUT.

Our analysis relied heavily upon the open source tool RapidSmith [19] developed at Brigham Young University. RapidSmith is a Java API that is able to both parse Xilinx design files and interface with part databases. In this project, RapidSmith was one of the key components that allowed us to do sensitivity analysis in both the logic and interconnect portion of the design.

*6.1. General Overview Analysis.* This subsection provides the reader with a general idea of the vulnerable cross section of our design. Specifically, it presents the design utilization, the bits in the bit-stream that were and were not tested, and the number of sensitive configuration bits found in the design.

Table 3 shows the device utilization of the DUT. While 23% of the BRAMs are instantiated less than half of them are actually used by the Linux kernel, making their effective utilization about 10%. The BRAMs not used by the Linux kernel are inserted by Xilinx's Embedded Development Kit (EDK) and are used to boot the PowerPC. Overall, the device utilization is between 10% and 12%.

Table 4 gives a summary of the bits that were and were not tested. The majority of these bits were BRAM and SRL16 bits. The number of sensitive configuration bits contributed by these resources was estimated as discussed in the next paragraph. We did not attempt to estimate the number of sensitive-masked bits or the IOB bits.

Table 5 presents the number of sensitive configuration bits found in the DUT and ICAP circuitry. The SRL16 bits were estimated by assuming that all content bits in the shift register would be sensitive. The same assumption was made for all BRAMs used by the Linux kernel (some were instantiated by EDK but not used by the kernel). The IOB bits were not tested, and no attempt was made to estimate the effect of their sensitivity on the device.

The Mean Time Between Failure (MTBF) is a common parameter used in reliability analysis. The equation for estimating the MTBF is presented in (1). The calculation requires that we know the configuration bit upset rate, $\lambda_{\text{bit}}$, and the number of sensitive configuration bits in the design, $N_{\text{bits}}$.

Table 4: Summary of bits tested/not tested.

| | | |
|---|---|---|
| Bits tested | 13,757,308 | **66**% |
| Bits not tested | 7,140,228 | **34**% |
| Mask file bits (not tested) | 821,636 | 4% |
| IOB bits (not tested) | 944,640 | 4% |
| BRAM content (not tested) | 5,373,952 | 25% |
| Total bits | 20,960,512 | **100**% |

Table 5: Summary of sensitive bits.

| | Sensitive bits | SRL16 bits | BRAM content | Total |
|---|---|---|---|---|
| DUT | 119,110 | 4,496 | 58,880 | 182,486 |
| ICAP | 1,329 | 0 | 0 | 1,329 |

The configuration bit upset rate is $\lambda_{\text{bit}} = 2.78 \times 10^{-7}$ upsets per day [20]. This failure rate is the same for commercial- and radiation-tolerant Virtex 4 devices. The difference between the commercial- and radiation-tolerant devices is the use of a thin epitaxial layer to remove single-event latch-up (SEL) and to significantly increase the total ionizing dose (TID) of the device.

$N_{\text{bits}}$ is the number of sensitive configuration bits found in the design. To estimate this value, we use the number of sensitive configuration bits we found during our fault injection experiment, 119,110. Additionally, upsets in BRAM and SRL16 content will also cause failures in the DUT. Since these components were not tested we estimate them by assuming that all bits in these elements are sensitive. We choose this approach because it provides a slightly pessimistic lower bound on the MTBF. The actual measured faults plus the estimated BRAM and SRL16 content bits brings $N_{\text{bits}}$ to 182,486.

Kernel failures can also be caused by upsets in the processor. Since our focus was on faults in the reconfigurable fabric, we did not perform any fault injection into the processor. Additionally, we do not know the number of state elements within the processor so we cannot form a pessimistic guess. Thus, our MTBF only indicates the average time between failures caused by the reconfigurable fabric. Using these numbers for $\lambda_{\text{bit}}$ and $N_{\text{bits}}$, we calculate the MTBF of our system due to faults induced by the reconfigurable fabric to be 19.7 days while in the IDLE operating mode:

$$\begin{aligned} \text{MTBF} &= \frac{1}{\lambda} = \frac{1}{\lambda_{\text{bit}} \times N_{\text{bits}}} \\ &= \frac{1}{2.78 \times 10^{-7}\,\text{upsets/day} \times 182,486\,\text{bits}} \\ &= 19.7\ \text{days}. \end{aligned} \tag{1}$$

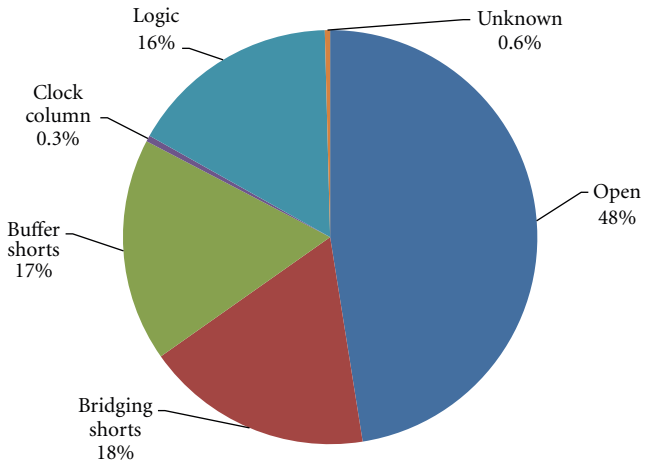


Figure 3: The distribution of sensitive configuration bits among utilized resources.

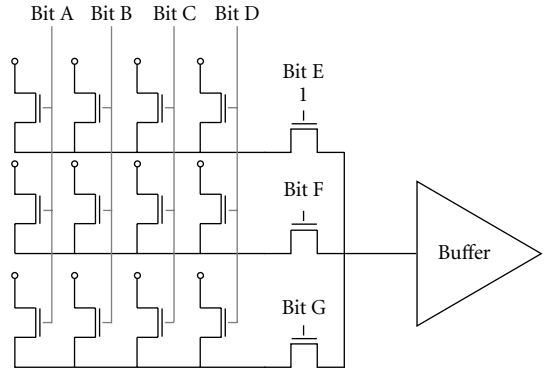


Figure 4: An example of a virtex 4 Routing Switch.

*6.2. FPGA Resource Analysis.* An FPGA resource analysis provides insight into which resources contained the sensitive configuration bits that cause system failure. This is done by mapping the sensitive bits analyzed in the previous section to the resources that they program. Understanding the resources involved in causing a design to fail could suggest improved resource-level mitigation approaches or future improvements to FPGA architectures.

Failures were placed into three sensitivity categories: Logic, Routing, and Unknown. The routing failures can be subdivided into three categories: open, bridging, and buffer failures. Figure 3 shows the distribution of the different types of failures identified in the design.

*6.2.1. Logic Failures.* Logic failures are the failures that occurred within the logic elements (slices) of the FPGA and accounted for 19% of all design failures. We were able to directly match most of the sensitive bits that caused these failures to components of slices that were specified as used in the design file. Some of the sensitive bits mapped to components of slices that were specified as unused in the design file; however, we often found that the unused (default) setting of the sensitive bit had relevance to the proper configuration of used slice components. For example, there is a mux at the input of the slice flip-flop set/reset line that allows the designer to choose between the inverted and noninverted version of the reset signal. When a flip-flop is instantiated in a design without a reset, an assumption is made by the bit stream generator about the default bitstream setting of the mux. If the bit that controls the mux is flipped by an SEU it will invert the default reset signal and hold the flip-flop in its reset state.

To compensate for this problem, we used the part database in RapidSmith to determine if an unused logic element's default setting could affect a used logic element. In about 10% of all logic sensitivities, an upset of the default setting was the cause of the failure.

*6.2.2. Routing Failures.* Routing failures accounted for 80% of all failure in the design. We now describe how routing failures occur, what their effects are, and how we identify them. To understand routing failures, one must first understand the structure of a programmable interconnect point (PIP) in the Virtex 4. While the structure of the Virtex 4 is not completely documented in the literature, the structure of a Virtex II pip is documented in [21, 22]. Our bit-stream studies lead us to believe a similar structure is used in the Virtex 4.

We conjecture that the Virtex 4 PIP (shown in Figure 4) is a two-level mux followed by a level restoring buffer. The select signals on this mux come from configuration bits A–G. Bits A–D are column bits while bits E–G are row bits. The small circles on the pass transistors represent connections from other wires in the switchbox. To properly pass a signal through the switch box, two configuration bits (a row bit and a column bit) must be set to "1" while all of the other bits must be set to "0".

For example, if we wanted to connect the wire coming into the top-left most pass transistor to the wire at the output of the PIP, we would set bits A and E to "1" and all the other bits to "0". Setting the other bits to "0" prevents contention within the PIP. By default, routing bits are set to "0". Since bits A and E are the bits that change from their default conditions we refer to them as primary bits.

Open failures are caused when an SEU sets one of the primary bits to a "0". This results in the disconnection of the input wire from the output wire. Since the PIP is now undriven the output will be driven high by the level restoring buffer.

To describe bridging and buffer failures, we have simplified the PIP in Figure 4 from a $3 \times 4$ PIP to the $2 \times 2$ PIP in Figures 5–8, where A and C are the primary bits in each PIP and net1 was originally passed through the PIP. Figure 5 shows the simplified PIP under normal (non-SEU) conditions. A lightning bolt on one of the configuration bits indicates that the bit has been changed to its current value by an SEU. The inside of the circles tells whether a buffer or net is driving the connection. The circle on the output describes a common function performed by the PIPs in the given configuration. Please note that these are common functions, not necessarily the function performed by every PIP in the given configuration. In fact, our data suggests that some invalid configurations of PIPs result in no failures at all.

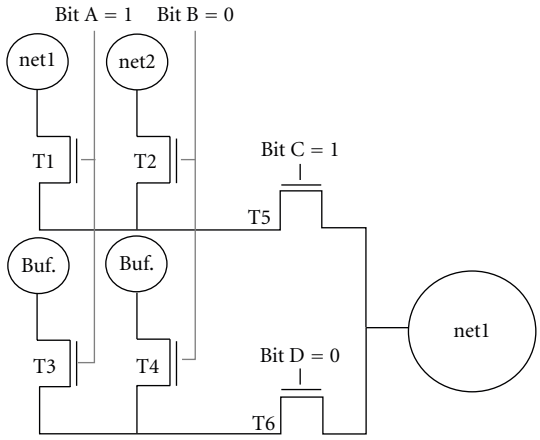


Figure 5: A simplified pip under normal operating conditions. No bits affected by SEUs.

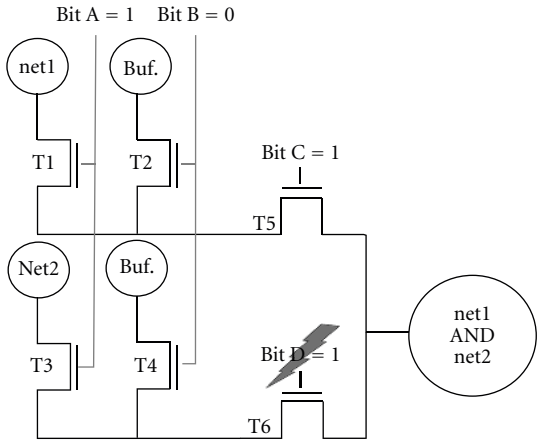


Figure 6: Example of a row bridging short. Bit D was affected by an SEU.

Figure 6 demonstrates a row bridging failure. The primary bit A allows net1 and net2 to drive the inputs of transistors T5 and T6, respectively. Since bit D was affected by an SEU, both row bits are active allowing both net1 and net2 to drive the output of the PIP causing a row bridging failure. Figure 7 demonstrates a column-bridging failure. In this case, net2 is driving the input of transistor T2, the SEU has effected bit B which allows both net1 and net2 to drive the input of T5 causing a column bridging short.

Figures 8 and 9 are examples of buffer failures and are exactly the same as row and column bridging failures with the exception that a buffer rather than a net is driving the connection created by the SEU.

In general, row-based buffer and bridging failures exhibit ANDing behavior. For a bridging failure this means that net1 and net2 are ANDed together. For a buffer failure this means that net1 is ANDed with a level restoring buffer, fortunately, this reduces to net1 being ANDed with a constant "1", which produces no effect on net1. While this ANDing behavior reduces the amount of row buffer failures, it does not always eliminate them. The majority of all row buffer failures

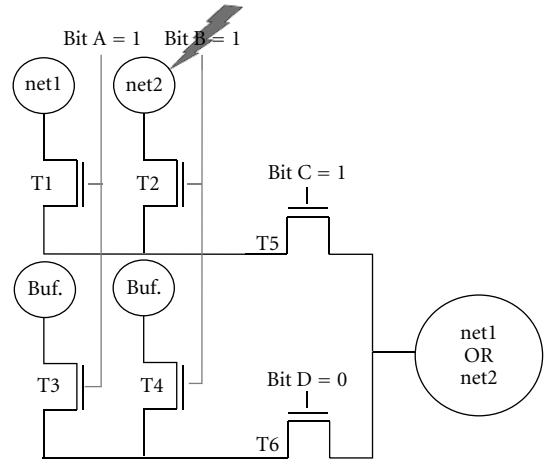


Figure 7: Example of a column bridging short. Bit B was affected by an SEU.

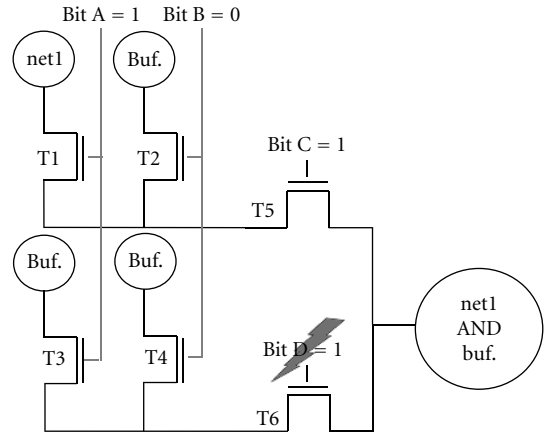


Figure 8: Example of a row buffer short. Bit D was affected by an SEU.

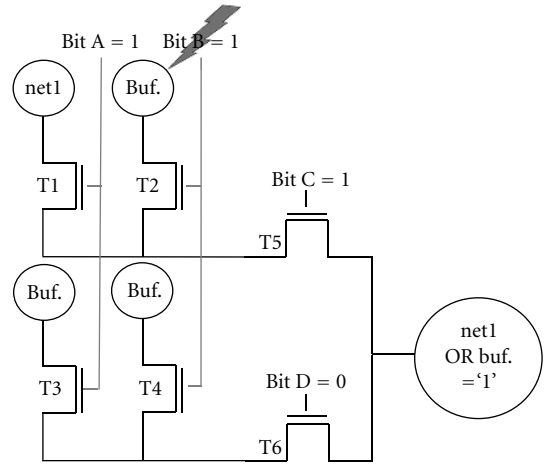


Figure 9: Example of a column buffer short. Bit B was affected by an SEU.

Table 6: Comparison of components.

| Module name | Primary bits | Interconnect bits | Logic bits | Bit-Area | # of sens. | Density metric |
|---|---|---|---|---|---|---|
| ICAP | 987 | 3,827 | 23 | 3,850 | 1,329 | 0.34519482 |
| IPLB | 2,439 | 11,035 | 2,276 | 13,311 | 3,641 | 0.27353317 |
| DPLB | 2,417 | 11,025 | 2,163 | 13,188 | 2,781 | 0.21087351 |
| PPC | 10,030 | 43,188 | 6,219 | 49,407 | 8,957 | 0.1812901 |
| DDR2 | 122,228 | 554,433 | 113,114 | 667,547 | 91,398 | 0.1369162 |
| GPIO | 917 | 4,522 | 1,688 | 6,210 | 850 | 0.136876 |
| SYSTEMRESET | 1,503 | 6,842 | 1,370 | 8,212 | 728 | 0.088650756 |
| CLOCKGEN | 11,966 | 48,194 | 562 | 48,756 | 4,145 | 0.08501518 |
| PLB | 18,121 | 78,556 | 10,864 | 89,420 | 3,449 | 0.038570788 |
| INTC | 2,236 | 10,642 | 2,932 | 13,574 | 365 | 0.026889643 |
| IIC | 11,558 | 53,990 | 13,562 | 67,552 | 927 | 0.0137227615 |
| UART | 2,633 | 12,563 | 3,425 | 15,988 | 203 | 0.012697022 |
| SystemAce | 3,093 | 14,348 | 3,808 | 18,156 | 113 | 0.006223838 |
| BRAMCNTRL | 11,525 | 50,616 | 4,099 | 54,715 | 25 | 4.5691308E-4 |
| JTAGDEBUG | 1,728 | 7,737 | 0 | 7,737 | 0 | 0.0 |
| MGTPROTECTOR | 13,504 | 64,784 | 0 | 64,784 | 0 | 0.0 |

in our fault injection test occurred in PIPs that are driven by the outputs of the slices. This is expected because the default settings of some logic elements in the slice is "0". Instead of the net being ANDed with a constant "1" which has no effect, the net is ANDed with a "0" which is always "0".

Column-based buffer and bridging failures typically result in ORing behavior. For a bridging failure, this means that net1 and net2 are ORed together. For a buffer failure, this means that net1 is ORed with a constant "1" which results in a constant "1" output for the PIP. In our fault injection test, column-based failures occurred twice as often as row-based failures for both bridging and buffer failures.

Once we understand the structure of PIPs and the ways that failures can occur, identifying these failures from fault injection results is straight forward. If the sensitive bit is a primary bit, then we know an open failure has occurred. If the sensitive bit is not a primary bit, the driver (buffer or net) of the connection created by the SEU will classify the failure as a buffer or bridging failure. We can further classify buffer and bridging as row or column failures by identifying whether the sensitive bit is a row bit or column bit.

Unknown failures accounted for less than 1% of all failures in the design. These were failures for which the sensitive bits that caused the failures could not be mapped to part of the design. Even though we do not know why these sensitive bits have caused design failure, we are confident in our fault injection results due to the fact that we were able to diagnose over 99% of the sensitive configuration bits that were found.

*6.3. User-Circuit Reliability Analysis.* The user-circuit reliability was analyzed on a module by module basis. In this analysis, the sensitive resources identified in the FPGA resource analysis are mapped to the modules that contain them. This information can be used to make decisions on the amount of redundancy used to increase the reliability of a system on a module by module basis. The primary method we have chosen in comparing modules is the sensitive bit density metric described in Section 5. This metric provides guidance on which modules would be most efficiently mitigated using redundancy techniques.

Table 6 provides the number of sensitive bits, bit area, and sensitive bit density metric for each component. The bit area is broken down into primary bits, interconnect bits (which also include primary bits), and logic bits. Recall from the previous subsection that primary bits are interconnect configuration bits that result in open failures. The Interconnect Bit column is the total of all interconnect configuration bits for the specified component. The logic bits column is the total number of bits used to configure the logic elements of the component. The bit area is the sum of the interconnect and logic bits for the component.

It is interesting to note that the majority of the bit area of each component is composed of nonprimary interconnect bits. The components with the highest densities of sensitive bits have far fewer sensitive configuration bits than the number of configuration bits specified in the interconnect bit area. This data seems to suggest that many shorting connections (caused by the faults in non-primary bit interconnect) do not affect the output of individual PIPs. This may suggest that our current method overestimates the actual sensitive bit area of a component; however, at the present time we do not know which non-primary configuration bits will cause errors and which ones will not; therefore, we must include all of these bits to avoid underestimating the bit area of the component. Further work would need to be done to conclusively show which bits will and will not cause shorting failures.

Since the ICAP component on the DUT is almost purely interconnect, it is comforting to see that the number of

sensitive ICAP bits found exceeds the number of primary bits. In fact, our test caught 959 of the 987 faults introduced into the primary bits of the ICAP. Twenty-six of the twenty-eight remaining primary bits exhibited the hiding behavior discussed in Section 5. The other two bits exhibited masking behavior. All 28 bits belonged to a state machine that initializes the ICAP immediately after configuration. After initialing the ICAP the nets in the state machine hold values of "1" that do not ever change. A fault in one of the primary bits of a PIP causes the PIP's output to be pulled high leaving the signal at its correct value and exhibiting hiding behavior. The fact that all the possible open failures within the ICAP were caught demonstrates that our tests of the ICAP during fault injection were sufficient to catch errors on all the interconnect of the ICAP module.

The first group of components we discuss are the modules that are on the main processor data path. These components occupy 4 of the first 5 rows of Table 6. These components are the IPLB, DPLB, PPC, and DDR2. The IPLB and DPLB are the connections between the PowerPC and the DDR2 memory controller. The PPC is the module that is used to instantiate PowerPC processor. The PPC defines the connectivity to the IPLB, DPLB, processor local bus (PLB), ground and VCC connections. The DDR2 is a multiported memory controller that has direct connections to the PowerPC through the IPLB and DPLB and also is connected through another port to the PLB.

One thing to notice about Table 6 is that the DDR2 contained 85% of the sensitive bits found in the fault injection test but did not have the highest sensitive bit density metric. The IPLB and DPLB each contained only 2% of the sensitive bits but had the highest density metrics. This suggests that using redundancy techniques on the IPLB and DPLB will result in a more efficient use of area than using redundancy techniques on the DDR2.

The reason the DDR2 has a lower metric than the IPLB and DPLB is because of the second port connected to the PLB. The second port adds more bit-area to the DDR2 component but does not add any sensitive configuration bits because the port is essentially unused. A solution to this problem would be to examine the DDR2 module and determine the sensitive-bit density metric of the port that is used by the processor. The metric is then likely to suggest that using redundancy techniques is likely more efficient for that one port. We also notice that the GPIO module, which the processor uses to respond to test requests also has a high sensitive bit density metric. The reader is referred to Table 6 for the remaining results.

We should also mention that the sensitive bit density metric may change under different operation modes of the system. Our results have only presented the metric from one operation mode. It is possible that the module with the highest metric may change throughout the operation of the design. This means that there are possible gains from using different mitigation techniques as the sensitive bit density changes during the run-time of the design. A possible way to accomplish this would be to use the dynamic partial reconfiguration ability of FPGAs.

## 7. Conclusion

A reliable Linux OS can be a useful tool on an FPGA-embedded system. Fault injection testing is an important first step in testing the reliability of FPGA-Embedded Linux Systems. Our test-bed provides an effective platform for fault injection and other useful experiments investigating the low-level details of the FPGA. Using different analyses in fault injection may help in identifying the lowest cost SEU mitigation techniques for FPGA-Embedded Linux Systems and provide additional confidence in fault injection results.

Our experiment showed that our Linux FPGA-embedded system was sensitive to 182,515 bits which gave our unmitigated system an MTBF of 19.7 days. We were able to match more than 99% of all failures to design utilized FPGA resources. We also found that in our system routing failures account for 83% of all design failures, while logic failures accounted for the other 16%.

In our FPGA resource analysis, we described the effects and identification of open, bridging, and buffer failures. We found that bridging and buffer failures often impose logic-like behavior on the net that was affected by the failure. Some of these logic-like behaviors affected the operation of the circuit while others did not.

We also suggested the use of the terms hiding and kernel resiliency to describe phenomena in which faults injected into the user circuit did not cause kernel failure. We even found an example of hiding in our ICAP circuitry that demonstrates the phenomena actually exists. Although we did not find an example of kernel resiliency, there are ways it could be evaluated. For example, by performing fault injection testing on the instruction and data memory of the Linux kernel, we may be able to draw correlations between the faults that cause kernel failure and the faults that do not.

In our user-circuit analysis, we showed that modules could be compared using the sensitive bit density metric. This metric helps determine which modules can be mitigated most efficiently using redundancy techniques such as TMR and PTMR. The sensitive bit density metric also indicated that using the same mitigation technique for the whole design may not lead to the most efficient use of FPGA area. Additionally, since the metric may change for a module during the operation of the circuit, it may be profitable to change the mitigation method of a module during run-time using dynamic partial reconfiguration.

## Acknowledgments

The authors would like to express their thanks to the Information Sciences Institute-East (ISI) for providing the ML410 boards and custom connector board. They would especially like to thank Neil Steiner for his assistance in the initial stages of their project.

## References

[1] B. Ramesh, T. Bretschneider, and I. Mcloughlin, "Embedded linux platform for a fault tolerant space based parallel

computer," in *Proceedings of the Real-Time Linux Workshop*, pp. 39–46, 2004.

[2] D. S. Katz, "Application-based fault tolerance for spaceborne applications," 2004, http://hdl.handle.net/2014/10574.

[3] M. Caffrey, "A space-based reconfigurable radio," in *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA '02)*, T. P. Plaks and P. M. Athanas, Eds., pp. 49–53, CSREA Press, June 2002.

[4] M. Caffrey, K. Morgan, D. Roussel-Dupre et al., "On-orbit flight results from the reconfigurable cibola flight experiment satellite (CFESat)," in *Proceedings of the 17th IEEE Symposium on Field Programmable Custom Computing Machines (FCCM '09)*, pp. 3–10, April 2009.

[5] M. Caffrey, K. Katko, and A. Nelson, "The cibola flight experiment," in *Proceedings of the 23rd Annual Small Satellite Conference*, August 2009.

[6] C. Carmichael, M. Caffrey, and A. Salazar, "Correcting single-event upsets through virtex partial configuration," *Xilinx Application Notes*, vol. 1.0, 2000.

[7] F. L. Kastensmidt, L. Sterpone, L. Carro, and M. S. Reorda, "On the optimal design of triple modular redundancy logic for sram-based fpgas," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '05)*, pp. 1290–1295, IEEE Computer Society, Washington, DC, USA, 2005.

[8] S. Baloch, T. Arslan, and A. Stoica, ""Probability based partial triple modular redundancy technique for reconfigurable architectures," in *Proceedings of the IEEE Aerospace Conference*, p. 7, 2006.

[9] D. L. McMurtrey, *Using deplication with compare for on-line error detection in FPGA-based designs*, Ph.D. dissertation, Brigham Young University, Provo, Utah, USA, 2006.

[10] F. Lima, C. Carmichael, J. Fabula, R. Padovani, and R. Reis, "A fault injection analysis of virtex fpga tmr design methodology," in *Proceedings of the 6th European Conference on Radiation and Its Effects on Components and Systems*, pp. 275–282, 2001.

[11] J. Monson, M. Wirthlin, and B. Hutchings, "Fault injection results of linux operating on an fpga embedded platform," in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs*, pp. 37–42, 2010.

[12] U. Legat, A. Biasizzo, and F. Novak, "Automated SEU fault emulation using partial FPGA reconfiguration," in *Proceedings of the 13th IEEE International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS '10)*, pp. 24–27, 2010.

[13] M. S. Reorda, L. Sterpone, M. Violante, M. Portela-Garcia, C. Lopez-Ongil, and L. Entrena, "Fault injection-based reliability evaluation of SoPCs," in *Proceedings of the 11th IEEE European Test Symposium (ETS '06)*, pp. 75–82, 2006.

[14] P. Civera, L. Macchiarulo, M. Rebaudengo, M. S. Reorda, and M. Violante, "FPGA-based fault injection for microprocessor systems," in *Proceedings of the 10th Asian Test Symposium*, pp. 304–309, November 2001.

[15] L. Sterpone and M. Violante, "An analysis of SEU effects in embedded operating systems for Real-Time applications," in *Proceedings of the IEEE International Symposium on Industrial Electronics (ISIE '07)*, pp. 3345–3349, 2007.

[16] 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 I, pp. 2147–2157, 2003.

[17] L. Sterpone and M. Violante, "Static and dynamic analysis of SEU effects in SRAM-based FPGAs," in *Proceedings of the 12th IEEE European Test Symposium (ETS '07)*, pp. 159–164, May 2007.

[18] L. Sterpone and M. Violante, "A new analytical approach to estimate the effects of SEUs in TMR architectures implemented through SRAM-based FPGAs," *IEEE Transactions on Nuclear Science*, vol. 52, no. 6, pp. 2217–2223, 2005.

[19] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, and B. Hutchings, "HMFlow: accelerating FPGA compilation with hard macros for rapid prototyping," in *Proceedings of the 19th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM '11)*, May 2011.

[20] G. Allen, "Virtex-4VQ dynamic and mitigated single event upset characterization summary," 2009, http://hdl.handle.net/2014/41104.

[21] C. Beckhoff, D. Koch, and J. Torresen, "Short-circuits on fpgas caused by partial runtime reconfiguration," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '10)*, pp. 596–601, 2010.

[22] S. Srinivasan, A. Gayasen, and N. Vijaykrishnan, "Leakage control in FPGA routing fabric," in *Proceedings of Conference on Asia South Pacific Design Automation (ASP-DAC'05)*, vol. 1, pp. 661–664z, 2005.