DrSEUs: A Dynamic Robust Single-Event Upset Simulator

Edward Carlisle, Nicholas Wulf, James MacKinnon, Alan George
NSF CHREC Center, ECE Department, University of Florida
327 Larsen Hall, 968 Center Drive, Gainesville, FL 32611
352-392-5225
{carlisle,wulf,mackinnon,george}@chrec.org

Abstract—**This paper presents DrSEUs (Dynamic robust Single-Event Upset simulator), a novel fault injector that uses the Simics full-system simulator. Fault-injection testing enables the use of commercial off-the-shelf (COTS) processors in space, which are susceptible to radiation-induced faults but are desirable due to the lower cost and higher performance of COTS devices. The de facto standard for fault injection is radiation-beam testing, which is often prohibitively expensive and time-consuming. Our methodology provides a means to iteratively decrease design vulnerabilities through rapid fault injection prior to beam testing. Additionally, our methodology can supplement beam-test results by targeting injections at individual components of interest that are difficult to isolate in beam tests. Our fault-injection mechanism uses simulation checkpoints, allowing DrSEUs to target a wide range of system components for injection. The deterministic nature of Simics checkpoints enables the repeatability of injection results and the monitoring of latent faults propagating through the system. We demonstrate the injection capabilities and analysis features of DrSEUs by presenting fault-injection results for an image-processing application.**

TABLE OF CONTENTS

1. INTRODUCTION

Space systems typically employ radiation-hardened processors to maintain high levels of reliability in the harsh environment of space. However, lower reliability commercial processors bring many advantages that appeal to space-system designers, including increased performance, power efficiency, and lower cost. Fault-injection testing is required to study the dependability of these components before launch. The de facto standard of fault injection for space-computing devices is radiation-beam testing, since beam testing irradiates components in a manner similar to space radiation. Unfortunately, radiation-beam testing is an expensive and time-consuming process. To minimize cost, designers often use alternative forms of fault-injection to iteratively analyze and improve their designs before arriving at the beam.

978-1-4673-7676-1/16/$31.00 ©2016 IEEE

Designers employ many forms of fault injection, including hardware-based, software-based, and simulation-based injection, where each type comes with tradeoffs. Hardware-based fault injection, including radiation-beam testing, is the most representative of the harsh environment of space, but risks causing permanent damage to the device under test (DUT). Software-based fault injection does not risk any damage to the DUT, but requires altering the target system's software in a way that may undesirably affect how faults manifest as errors. Simulation-based fault injection is unique because simulations offer the most visibility into and control over the DUT. However, the accuracy of results depends on the fidelity of the simulation models. We chose the Simics toolset, detailed in [1] and [2], from Wind River for this research because of Simics' extensibility and capability to run full target-application binaries on simulated processors. We can apply the methodology presented in this paper to any processor model in the Simics catalog (and possibly to other full-system simulators); however the target processor for this research is the Freescale P2020, a dual-core, PowerPC-based processor studied by JPL [3] and featured in the Proton400k single-board space computer from Space Micro [4].

Our methodology relies on Simics' checkpointing feature, which can save and restore the entire state of the simulated system at any point during execution. Our methodology uses these checkpoints to perform injections and compare the injected system state to a previously established gold system state (i.e., without fault injection). Since these checkpoints contain the entire state of the system, we can target a wide range of components for injection. We perform fault injections on these checkpoints without any interaction with Simics, such that the target system, and even Simics, is unaware of the injection of faults. Furthermore, these checkpoints ensure that all fault-injection runs are isolated from one another, facilitating error diagnosis and classification. These checkpoints also allow for the detection of latent faults that exist in the system that may not affect the target application's execution but can manifest as errors in subsequent operations.

The remainder of this paper is organized as follows. Section 2 covers related work, including a survey of existing fault injectors. In Section 3, we discuss the scope of our methodology and the unique features of our fault injector. DrSEUs, the Dynamic robust Single-Event Upset simulator, includes analysis features that neatly organize and display campaign results in tables and charts enabling advanced analysis. We present our fault-injection methodology in Section 4. Section 5 presents the experimental setup for our

fault-injection campaign that we performed on an edge-detection, image-processing application. In Section 6, we present our analysis of the fault-injection campaign, which demonstrates the depth of analysis achievable through manipulating the variety of information DrSEUs collects. We also study the effects of latent faults by performing additional computation when latent faults exist in the system. These detailed results are only possible through simulation-based fault injection, as controlling the number and location of single-event upsets is not possible with radiation-beam testing nor do hardware testbeds offer the visibility required for the detection of latent faults. We present our conclusions in Section 7 and discuss future work for this research.

2. RELATED WORK

The effects of radiation on electronics are an important area of research for space computing. Karnik et al. present a study on the interactions of radiation particles with VLSI circuits in [5]. In order to simulate these effects in the lab and understand the impact of these effects on system reliability, researchers can perform fault-injection testing. Quinn et al. explain the importance of fault-injection testing as a means to assess a system's reliability in the presence of harmful radiation in [6].

Researchers have classified and compared several methods for performing fault injections, as in [7] and [8]. There are four primary categories of fault injection, each with unique advantages and disadvantages, including hardware-based, software-based, simulation-based, and emulation-based.

Hardware-based Fault Injection

Hardware-based fault injection is composed of two subcategories, fault injection with or without physical contact, as discussed in [7] and [8]. Hardware-based fault injection without physical contact is most similar to what a device would experience in a space environment. This method uses either radiation or electromagnetic interference to cause faults in the DUT. Hardware-based fault injection with physical contact uses either active probes or socket insertion to introduce voltage or current changes to the DUT. This category of fault injection can simulate open faults, short circuits, bit flips, spurious current, power surges, or stuck-at faults. Drawbacks to hardware-based fault injection include possible damage to the DUT and the necessity to modify the DUT.

Software-based Fault Injection

Software-based fault injection is popular since it is portable, does not usually require any hardware modifications to the DUT, and does not risk damage to the DUT. However, software-based fault injection also comes with disadvantages, for example not being able to inject faults into places that are not accessible through software, such as caches. Software-based fault injectors also introduce the possibility of disturbing the processing workload in unintended ways. For example, adding additional software required for performing injection may alter the scheduling and timing of system tasks, as discussed in [7].

The Simple Portable Fault Injector (SPFI), is a software-based fault injection tool presented in [9]. SPFI uses the GNU Debugger (GDB) to pause execution and randomly inject single-bit flips into CPU registers and memory values during the execution of a targeted application. However, the use of GDB as the injection mechanism limits injection scope to the targeted application. Therefore, it is not possible to study the effects of injected faults on the remainder of the system (e.g., operating system, device drivers).

JPL's Implementation of a Fault Injector (JIFI) is similar to SPFI, but uses ptrace (the Unix system call used by GDB to control other processes) instead of GDB to inject faults, offering lower-level access to the system [10]. JIFI requires modification of the targeted application to include specific function calls that perform fault injections.

Many other software-based fault injectors exploit the interrupt-handling capabilities of modern processors to trigger code that performs injections. These are commonly referred to as code emulating upsets (CEU) and are studied in [11], [12], [13], and [14].

Simulation-based Fault Injection

Simulations also serve as a testbed for fault injection. In fact, injecting faults in a simulation model has some advantages over injection in a physical system. Simulations operate at different levels of abstraction, which allows the use of multiple fault models. Due to the tight integration of fault-injection mechanisms and system-simulation models, fault injections become transparent from the target system's point of view. Simulations also provide the most visibility into and control over both the target system and the fault-injection mechanism.

Simulation-based fault injectors can use hardware description language (HDL) models of targeted devices as in [14]. Unfortunately, obtaining HDL models of commercial processors is often difficult. Instead, researchers can perform fault injections using full-system simulators, which provide all of the functionality required to run full software stacks for the targeted device. Velazco et al. [15] use d3sim, a DSP simulator, to inject faults into a simulation of the DSP32C. Other studies have used Simics, another full-system simulator detailed in [1] and [2], as a testbed for fault injection. For example, Bastien implements the Saboteur Module [16] as a Simics module to inject faults in the simulation of an x86 processor. This module is capable of injecting transient or permanent faults into CPU registers, memory or I/O data busses, and memory address busses. In [17], Chao et al. modified Sam, a chip multithreading (CMT) simulator from Sun Microsystems built atop Simics [18], in order to develop the Full system Simulator-based Fault Injection (FSFI) tool and perform testing on the UltraSPARC T2 processor.

Emulation-based Fault Injection

The final fault-injection category is emulation based. This category is similar to HDL-based simulation fault-injection. However, instead of using an HDL simulator to perform injections, the DUT is synthesized onto a field programmable gate array (FPGA) and augmented with fault-injection mechanisms. The study in [19] presents a methodology for emulation-based fault injection on both the MicroBlaze and Leon3.

Simics

Simics, detailed in [1] and [2], provides cycle-accurate simulations at the instruction-set level and includes simulation models for peripheral components, such as memory and interrupt controllers, PCI, Ethernet, etc. Simics allows for the execution of unmodified operating systems, firmware, device drivers, middleware, network stacks, and of course applications. The internal state of the processor, memory contents, and executing instructions are all exposed during simulation. Another important Simics feature for our methodology is the ability to save a checkpoint, containing the entire state of the system, which can be later loaded in Simics to continue the simulation from the same state.

3. DRSEUS OVERVIEW

Our fault-injection methodology benefits from the advantages of simulation-based fault injection. Most notably, the high level of visibility into the system provides a wide range of components to target for injection. This property also allows us to study the propagation of faults throughout the system as well as the effects of latent faults. Unlike most of the fault injectors surveyed in the previous section, DrSEUs does not require any modifications to the DUT, including both software and (simulated) hardware modifications. This property also extends to the simulator itself, since our fault-injection mechanism is external to Simics. In fact, our methodology is extendable to other full-system simulators, so long as the simulators provide the same level of simulation, allow access to the same interfaces to the DUT, and can save checkpoint files containing the state of the entire system at a given point in time.

Our methodology is not meant to replace radiation testing. Instead, it enables system designers to better prepare for radiation tests by iteratively improving their designs through fault-injection testing. This preparation allows designers to maximize the effectiveness of beam time, which is typically a limited resource. Our methodology also provides much greater control in targeting device components for injections than is possible with radiation beams. Therefore, when designers find individual components that are exceptionally sensitive to radiation-induced faults, our methodology allows them to supplement radiation-test results with targeted fault injections to study the effects on the system in greater detail.

DrSEUs simulates two of the most important single-event effects (SEE): single-event upsets (SEU) and single-event functional interrupts (SEFI). However, there are certain SEEs and other radiation effects that Simics, and consequently our fault injector, cannot accurately model. These include single-event transients (SET), single-event latchup (SEL), single-event burnout (SEB), single-event gate rupture (SEGR), and cumulative effects like total dose. While Simics can model SETs in certain components of the system (address and data buses), this functionality would require modifying the simulation model to include fault-injection modules. Our methodology aims to be completely transparent in order to avoid effects caused by the unintended consequences of modifying the DUT. As previously stated, we inject faults into checkpoints outside the context of Simics to achieve transparency. Other simulation-based, fault-injection techniques that use register-transfer level (RTL) models are capable of simulating SETs in a wider range of components (including combinatorial-logic networks). However, acquiring the RTL models for commercial processors is usually not possible. Simulating SELs, SEBs, and SEGRs would require much lower-level modeling akin to SPICE-based simulations. These lower-level simulations cannot offer the same magnitude of performance that Simics provides for system-level simulations.

In order to achieve high performance, DrSEUs can perform multiple injections in parallel to speed up campaign progress. Each injection instance uses a private instance of Simics to isolate the simulation. The experiment in this paper uses as host a quad-core Core i7 processor with eight threads to instantiate eight parallel instances of the fault injector in order to keep the host busy at all times. Our methodology can easily scale to computing clusters in order to perform massively parallel fault-injection campaigns. Our performance is limited to the number of CPU cores and available Simics licenses (each instance of Simics requires a license).

DrSEUs also includes many features to aid in fault-injection campaign analysis. One of these components is a web application that organizes results into tables and charts that are dynamically generated. We present some of these charts in the experiment section of this paper. A filtering capability is included to narrow down the data in the tables and charts to only the results of interest. Each result includes the associated injection data, DUT console output, Simics console output, and lists of latent faults located in registers, TLB entries, and memory blocks. Checkpoint regeneration is another powerful feature to aid in analysis. Due to the deterministic nature of Simics simulations, DrSEUs can regenerate injected system checkpoints and launch these checkpoints in Simics allowing further analysis of interesting results. This feature also allows designers to use Simics' advanced debugging capabilities to further study faults.

4. METHODOLOGY

Figure 1 shows the architecture of our fault injector. DrSEUs interfaces with Simics through standard in (STDIN) and out (STDOUT), allowing DrSEUs to control and monitor Simics. Simics connects the DUT's serial console to a pseudo-terminal on the host, mimicking the serial connection to a physical device, allowing DrSEUs to issue commands and monitor execution. Simics also forwards a network port from the host to the DUT, via a virtual Ethernet connection. This network connection allows DrSEUs to send and receive files, including application binaries and input/output files. If the simulated device does not include an Ethernet interface, Simics provides a SimicsFS kernel module that can mount the host's file system and transfer files. Finally, DrSEUs modifies checkpoint files, created by Simics, in order to perform fault injections. DrSEUs can then load these modified checkpoint files in Simics to continue the simulation from an injected state.

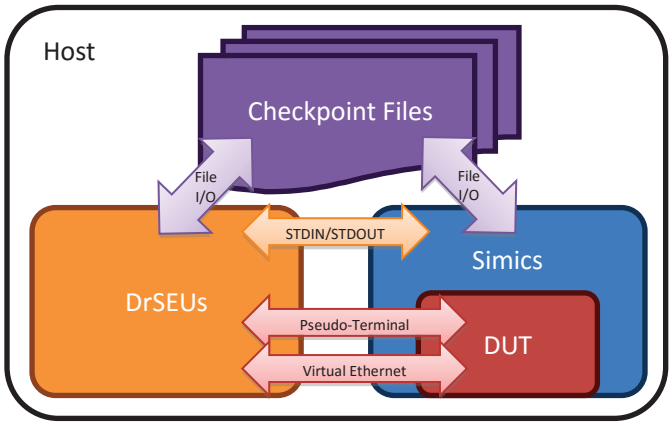

Figure 1. Fault injector architecture

Before performing any fault injections, we augment the targeted application with signal handlers that print easy-to-parse messages indicating which signal the application encounters. These signal handlers are not required but can facilitate error diagnosis and classification. Optionally, we can also augment the targeted application with messages that print statistics, such as the number of detected errors, so DrSEUs can include these statistics in the logged results.

Our methodology begins with the creation of a new fault-injection campaign, outlined in Figure 2. At this stage, DrSEUs starts Simics with a script that instantiates all components of the DUT and creates connections to the host. Once Simics has instantiated all components and connection, the DUT boots Linux. Next, DrSEUs uses SCP (Secure Copy, which copies files using the Secure Shell, or SSH, protocol) to transfer application binaries and input files to the DUT via Simics' forwarded network port.

After transferring the necessary files, DrSEUs times the execution of the targeted application. In this step, the DUT runs the targeted application and Simics measures the number of elapsed DUT clock cycles. Dividing the measured cycles by the desired number of gold checkpoints calculates the number of cycles that should separate the checkpoints. To create these gold checkpoints, DrSEUs halts the simulation and queues a command for the DUT to run the targeted application. Then, for each desired gold checkpoint, Simics advances the simulation the appropriate number of cycles and creates a checkpoint. Finally, after Simics creates the last checkpoint, DrSEUs resumes the simulation and uses SCP to transfer the output file to the host for later comparison.

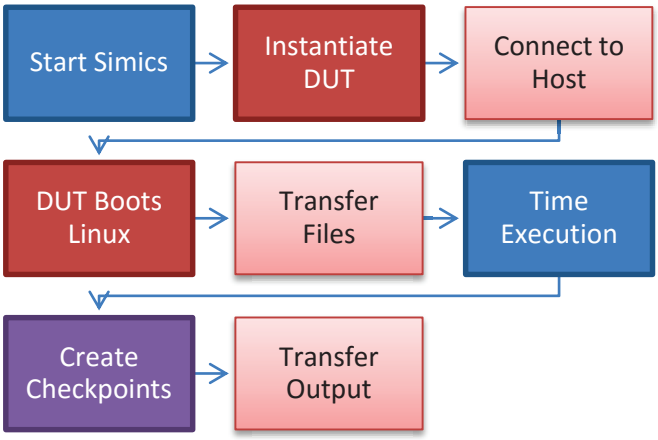

Figure 2. Campaign creation

Performing Fault Injections

We use the gold checkpoints as the entry point for our fault-injection mechanism. Checkpoints contain the entire state of the simulation for a given point in time, including memory contents, general-purpose register values, CPU control-register values (e.g. program counter), SoC peripheral component controller (e.g. Ethernet controller) register values, and translation-lookaside buffer (TLB) entries. DrSEUs targets all of the components contained on the processor for injection, excluding caches as they are not included by default in Simics' models. We plan to include cache models for fault injection in future work, as we discuss further in Section 7. DrSEUs does not target memory contents or other devices external to the processor for injection.

As shown in block 1 of Figure 3, DrSEUs begins each fault-injection iteration by randomly picking one of the gold checkpoints for injection (excluding the final checkpoint, as the application has already completed execution by this point) and copying all files associated with this checkpoint to the injection directory. By modifying a copy of a gold checkpoint for each iteration, DrSEUs achieves two desirable characteristics. First, this process provides isolation between iterations, facilitating error diagnosis and classification. Second, campaign progression is accelerated because Simics does not need to repeat the device instantiation and DUT boot process for each iteration.

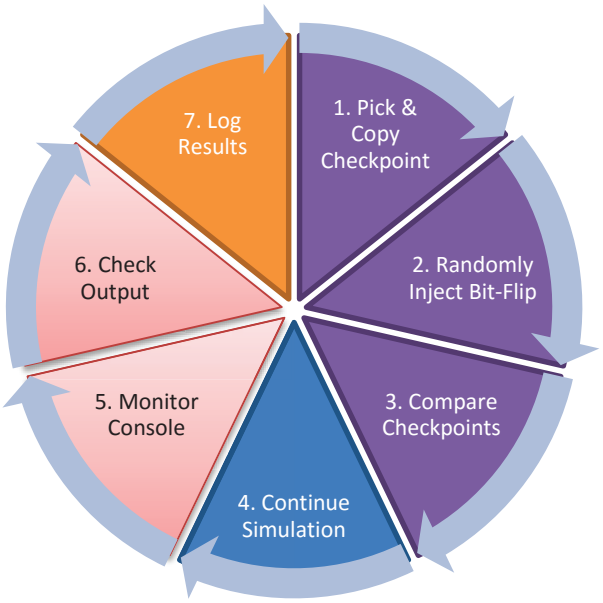

Figure 3. Fault injection

In block 2 of Figure 3, DrSEUs randomly chooses a register or TLB entry for injection and uses the number of bits each target contributes to the overall system to distribute injection probability. DrSEUs then injects a fault by flipping a random bit of the selected target in the copied checkpoint and loads the modified checkpoint in Simics. Because fault injections take place outside the context of Simics, the DUT (and even Simics) is unaware that an injection has occurred.

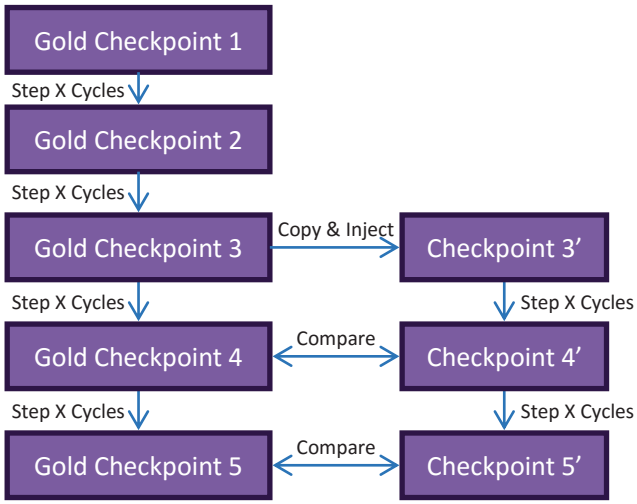

Figure 4. Injection and checkpoint comparison

Execution monitoring begins in block 3 of Figure 3. In this phase, Simics advances the simulation the appropriate number of cycles and saves the next checkpoint. DrSEUs then compares this checkpoint with the corresponding gold checkpoint in order to track the propagation of faults (in memory blocks, TLB entries, and register values) throughout the system. DrSEUs repeats this process, comparing each checkpoint with the equivalent gold checkpoint. Figure 4 demonstrates the injection and comparison process for an example campaign of 5 checkpoints, where DrSEUs chooses checkpoint 3 for injection. In the fault-injection campaign that we present in the next section, we skip checkpoint comparison until the final gold checkpoint in order to reduce the time required for execution monitoring. After comparison with the final gold checkpoint, DrSEUs continues the simulation and monitors the DUT's console, shown in blocks 4 and 5 of Figure 3, checking for messages related to execution errors (including signal handler messages and Linux kernel errors) while using a timeout to detect if the DUT is hanging. If the application completes without any execution errors, DrSEUs uses SCP to retrieve the output file for comparison with the gold output file (retrieved during campaign creation) to check for data errors, shown in block 6 of Figure 3. In the case where the application completes successfully yet DrSEUs detects faults when comparing checkpoints, DrSEUs runs the target application a second time to determine the impact of these latent faults on the system. Finally, DrSEUs logs all results for later analysis, as shown in block 7 of Figure 3.

DrSEUs can perform multiple fault-injection iterations in parallel to accelerate campaign progression. To achieve this, DrSEUs concurrently injects multiple checkpoints and launches each checkpoint in a separate instance of Simics. Each instance of Simics uses unique pseudo-terminals and host network ports for execution monitoring, preventing any cross communication between simulated DUTs.

5. Experiment Setup

In this section, we present our experimental setup for a case study, which we analyze in the next section. Our experiment includes a fault-injection campaign for a Simics simulation of Freescale's PowerPC-based P2020 running an edge-detection, image-processing application. The campaign uses 1000 checkpoints to provide a fine granularity for injection time and includes over 71,000 fault injections.

Injection & Result Descriptions

Figure 5 shows each component of the P2020 targeted for injection and the components' contribution to bits targeted for injection. These include: configuration, control, and status registers (CCSR); CPU control registers (i.e., program counter, memory-management assist registers, etc.); direct memory access (DMA) controller registers; e500 coherency module (ECM) registers; enhanced local-bus controller (ELBC) registers; enhanced secure digital host controller (ESDHC) registers; enhanced serial peripheral interface (ESPI) controller registers; enhanced three-speed Ethernet controller (ETSEC) registers; general-purpose input/output (GPIO) registers; general-purpose registers (GPR); I^2C module registers; L2 SRAM registers; memory controller (MC) registers; PCI express controller registers; programmable interrupt controller (PIC) registers; RapidIO controller registers; TLB entries; and universal serial bus (USB) controller registers.

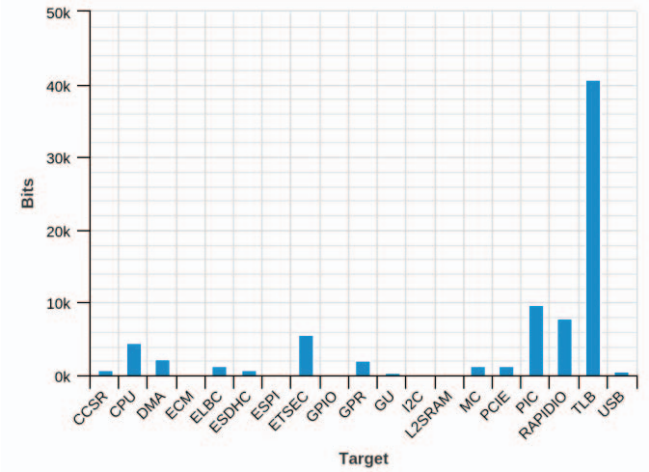

Figure 5. P2020 component contributions to bits targeted for injection

Table 1 lists and describes each of the result classifications for fault-injection iterations. In Section 6, we group the figures included in our analysis by outcome (for detail) or by category (for conciseness).

Application Description

Edge detection, and convolution in general, is an essential part of high-level algorithms in machine vision. Feature detection uses convolution-based edge detection to find corners or points of interest, which computer-vision algorithms can use to autonomously determine image quality. Spacecraft sensors can generate large amounts of raw data, yet downlink capabilities are limited. Therefore, it is necessary to perform image processing in situ and intelligently transmit only interesting images in order to conserve bandwidth.

In our case study, we implement convolution using Fourier transforms instead of the direct form of convolution. FFT convolution is favorable because processing in the frequency domain is less computationally intensive for typical space-camera image resolutions and large kernels. This optimization consists of performing the image processing in the frequency domain (by multiplying the transformed kernel and image) and then computing the inverse FFT on the result, as shown in Figure 6.

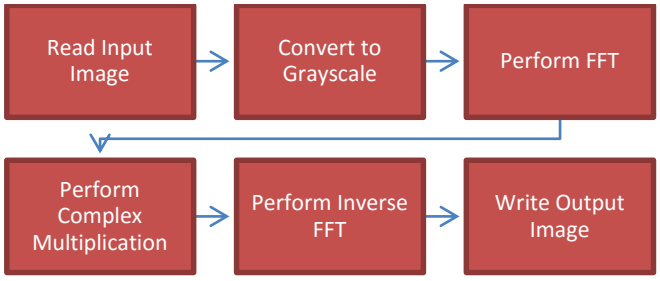

Figure 6. Stages in image-processing application

Table 1. Fault injection classifications

Category	Outcome	Description
No Error	Latent Faults	Application executed successfully, but DrSEUs detected faults when comparing checkpoints
	Masked Faults	Application executed successfully, and DrSEUs did not detect any faults when comparing checkpoints
	Persistent Faults	Application executed successfully, and the only fault DrSEUs detected when comparing checkpoints was the originally injected fault
Data Error	Detected Data Error	Application completed execution, but reported data errors
	Silent Data Error	Application completed execution, but output file did not match gold output file
Execution Error	Hanging	Application failed to complete execution and DUT became unresponsive
	Illegal Instruction	Application failed to complete execution due to an illegal instruction (not reported by signal handler)
	Kernel Error	Application failed to complete execution due to Linux kernel error
	Segmentation Fault	Application failed to complete execution due to a segmentation fault (not reported by signal handler)
	Signal SIGILL	Application failed to complete execution and signal handler reported SIGILL was raised
	Signal SIGIOT	Application failed to complete execution and signal handler reported SIGIOT was raised
	Signal SIGSEGV	Application failed to complete execution and signal handler reported SIGSEGV was raised
	Signal SIGTRAP	Application failed to complete execution and signal handler reported SIGTRAP was raised
Simics Error	Address Not Mapped	Simulation halted due to unmapped memory address
	Dropping Memop	Simulation halted due to unmapped memory address
SCP Error	Missing Output	Application completed execution, but SCP failed to retrieve the output file from the DUT
Post-Execution Error	*	Latent fault outcome where a subsequent run of the application failed

6. Analysis and Results

This section demonstrates the advanced analysis made possible by DrSEUs. We collect a variety of information during fault-injection campaigns, which allows us to organize the results in different ways to gain unique insights into the system and our targeted application.

Randomness of Injections

Before beginning our analysis, we demonstrate the randomness of injections. Figure 7 shows that DrSEUs evenly distributes the randomly injected faults over all bits targeted for injection, as these injection ratios closely match the bit ratios shown in Figure 5.

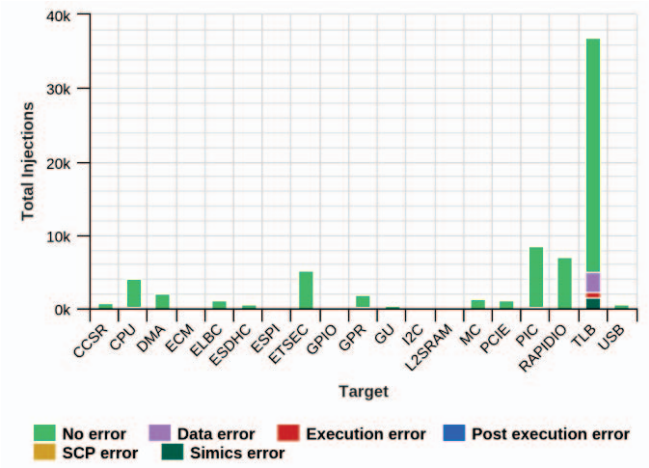

Figure 7. Total injections performed for each target, grouped by category

Due to the great disparity in sizes among the injection targets, Figure 7 impedes the visualization of outcomes that result from injections into some of the targets. To facilitate analysis, Figure 8 reorganizes the data of Figure 7 to show the percentage of outcomes for each target.

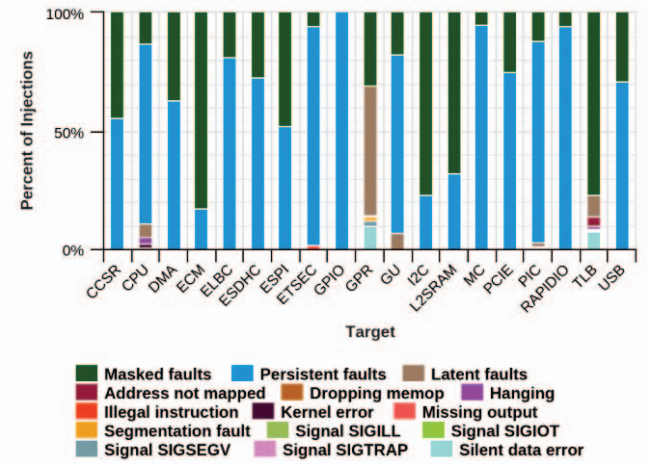

Figure 8. Percentage of injections for each target, grouped by outcome

Detail of Results

We can also view the results for each register within a selected target. For example, Figure 9 shows results for injections in each general-purpose register. By compiling the results in this manner, we can perform further analysis to correlate the vulnerability and usage of each general-purpose register. For example, the PowerPC architecture commonly uses r1 the stack pointer, which explains the occurrence of SIGSEGV signals (denoting a segmentation fault). We can inspect general-purpose register usage by analyzing a targeted application's assembly-language representation, which a disassembler can facilitate.

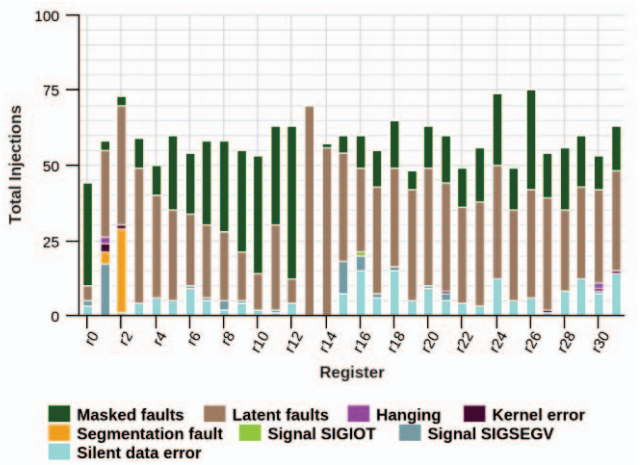

Figure 9. Total injections performed for each general-purpose register, grouped by outcome

Overview of Result Categories

Figure 10 shows an overview for the injection campaign, with results grouped by category. As shown in the figure, most injected faults result in the no error category and do not affect the application's execution or output.

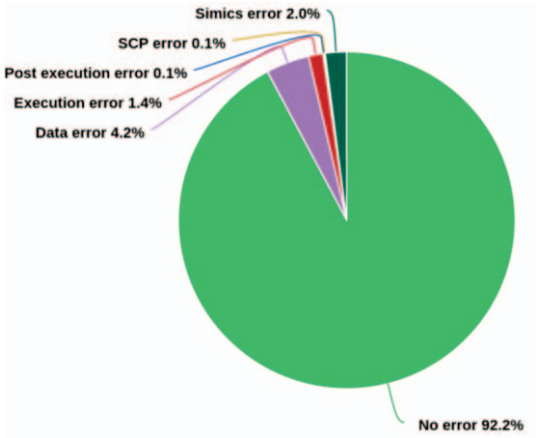

Figure 10. Campaign overview of result categories

No Error—While the results in the no error category did not affect the targeted application's execution or output, Figure 11 shows half of these results stem from the latent and persistent fault outcomes, indicating the injected fault still affects the system. In these cases, we perform a second execution of the application to determine the effects of these latent faults on subsequent operations. Although the targeted application is unaffected, we note that other applications, or possibly the Linux kernel, can later suffer from an error. We

explore this possibility further in the analysis of the post-execution error category, shown in Figure 16.

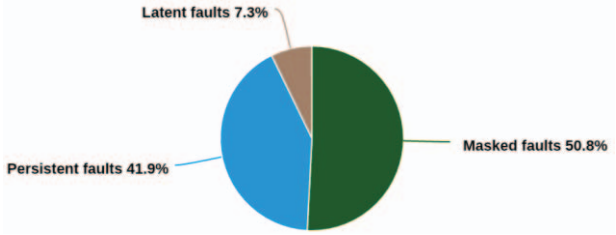

Figure 11. Outcomes in no error category

Data Error—Data error is the most common category that results in errors. All of the data errors for this application are silent data errors, as this application did not include any form of fault-tolerance to detect data errors. Figure 12 plots the severity of data errors for each target, demonstrating that injections into the Ethernet controller are the most destructive errors for the output data. These destructive errors are attributed to the fact that we use Ethernet to retrieve the output file from the DUT. Fortunately, Figure 13 shows the number of injections into the Ethernet controller resulting in data errors is quite low.

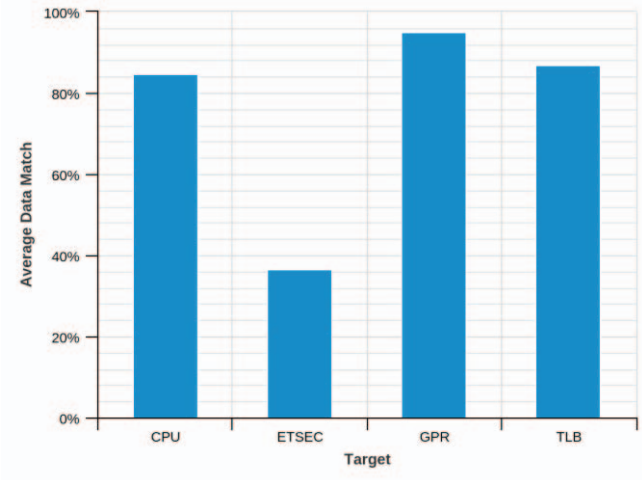

Figure 12. Average data match percentage for targets, filtered for data errors

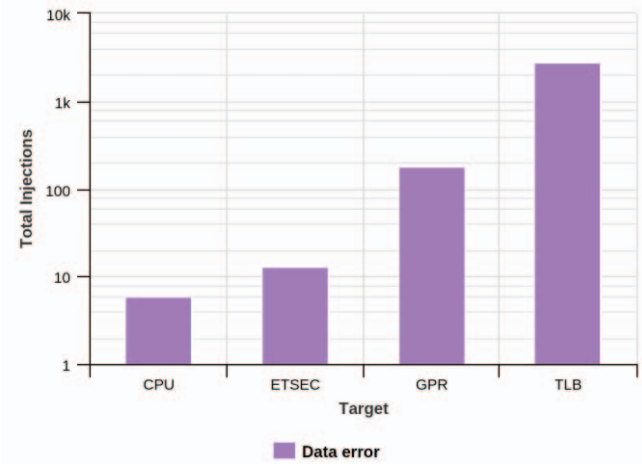

Figure 13. Total injections for targets, filtered for data errors

Execution Error—The next most common result category is execution error. Figure 14 shows the breakdown of outcomes that fall under this category. The most common outcome in the execution error category is hanging, where the device simply becomes unresponsive. After these outcomes are segmentation faults (caught and uncaught), followed by illegal instructions (caught and uncaught).

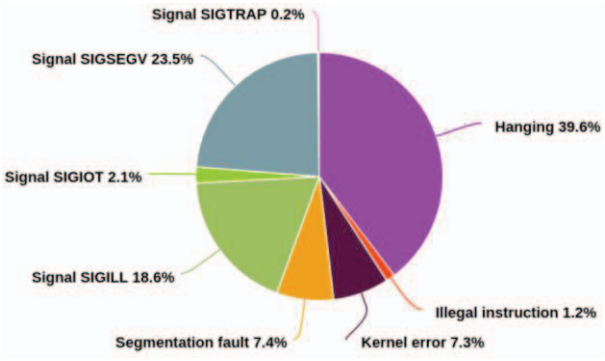

Figure 14. Outcomes in execution error category

Simics Error—Simics error is the next most common category. Both of the outcomes that fall under this category result from the system attempting to access a memory address that is unmapped. In a physical system, this would result in an execution error. However, in Simics, these actions prevent the simulation from advancing. Unsurprisingly, all faults that result in Simics errors are due to TLB injections.

SCP Error—We classify an iteration in the SCP error category when an error occurs while trying to retrieve the output file from the DUT (over SCP). We can filter the chart in Figure 7 to only display results in the SCP error category resulting in Figure 15, which clearly shows that SCP errors most commonly result from injections into the Ethernet controller, which is not surprising given that SCP communication takes place over Ethernet.

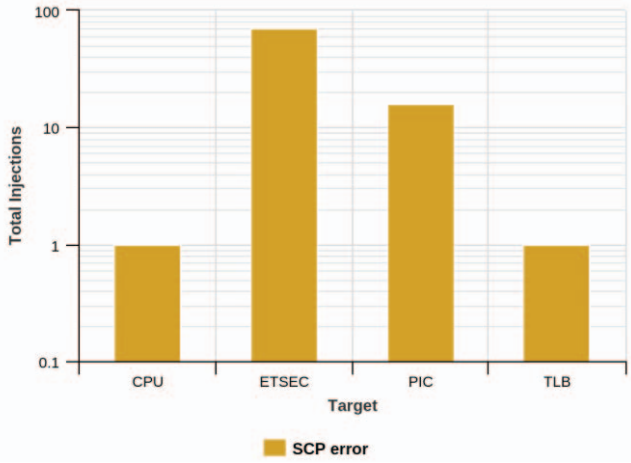

Figure 15. Injection targets resulting in SCP errors

Post-Execution Error—The final injection result category is post-execution error. We classify an iteration in this category if, after the target application executes successfully, DrSEUs detects latent or persistent faults in the system and a second execution of the application fails. Figure 16 shows the breakdown of outcomes in the post-execution error category. The figure clearly shows that the faults in this category cause errors at the system level and result in kernel errors and system unresponsiveness, leading us to the hypothesis that other results in the latent- and persistent-fault categories can still lead to system errors even if the injected faults did not have an immediate effect on the target application.

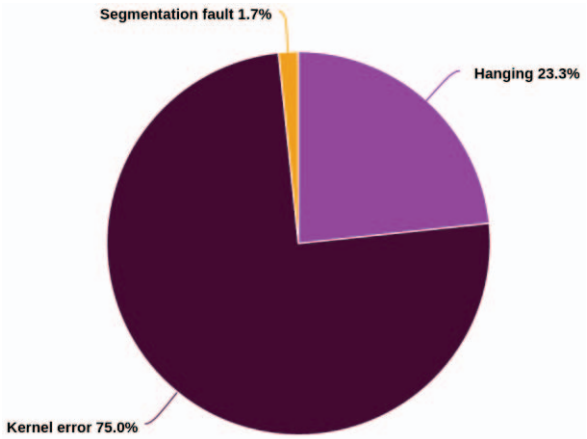

Figure 16. Outcomes in post-execution error category

Injection Time

Plotting the results over injection time (or checkpoint number) presents a unique view into the vulnerability of the target application. Figure 17 applies a moving average filter to the results to increase the clarity of trends showing that there are three distinct phases of our application that are more vulnerable than the rest of the application. Monitoring the devices console output while creating the gold checkpoints in the campaign creation process allows us to correlate ranges of checkpoints with processing stages. Table 2 shows this correlation. It is now apparent that the forward and inverse FFT phases of the application are by far the most vulnerable, as the first two areas of elevated errors occur during the forward FFT phase and the third are of elevated errors occurs during the inverse FFT phase.

Table 2. Checkpoints for each application stage

Execution Phase	Checkpoint Range
Read input image	0-1
FFT preparation	1-32
Forward FFT	32-605
Complex multiplication	605-643
Inverse FFT	643-964
Save output image	964-999

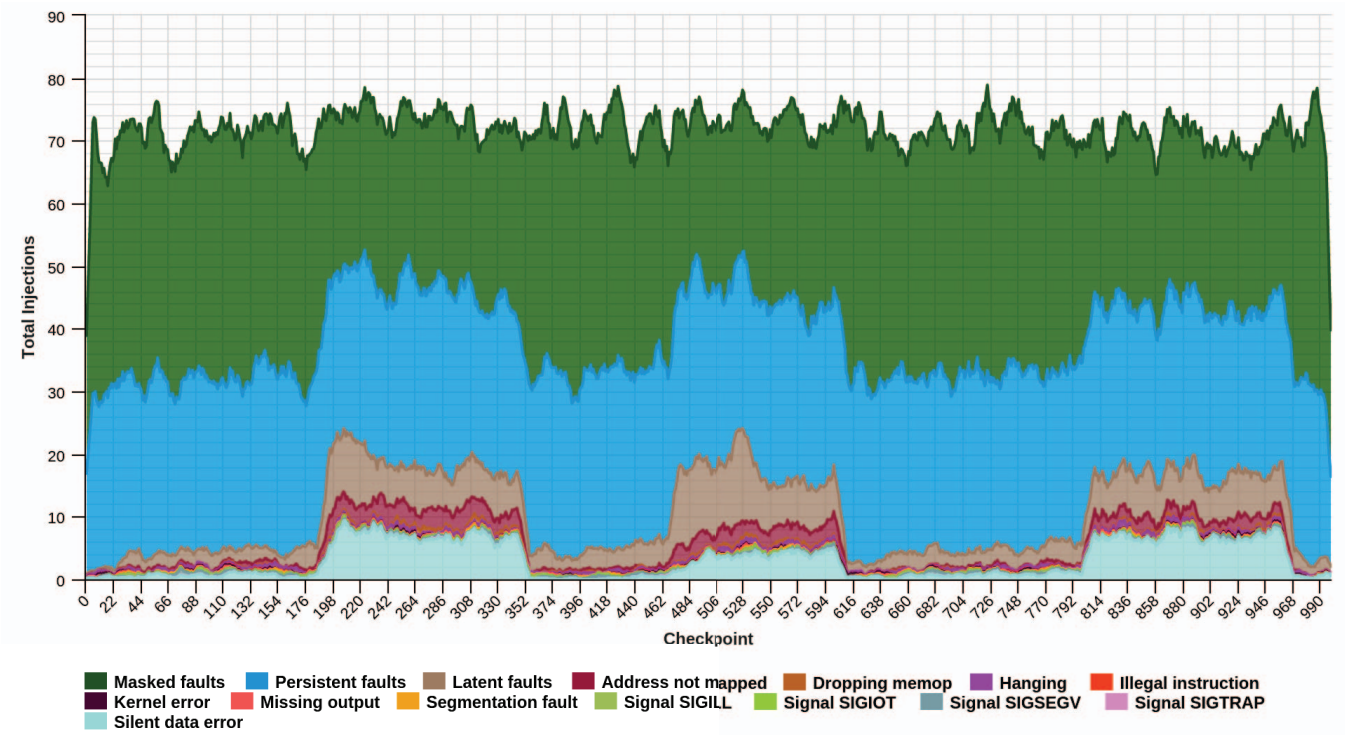

Figure 17. Injections over time

Latent Faults

The unique ability of our fault injector to detect latent faults in the system allows us to measure the spread of faults throughout the system. Figure 18 shows the average number (across all results) of registers and memory blocks that an injected fault corrupts. We can see that only a few injection targets contribute significantly to the propagation of faults, and these are CPU, GPR, and TLB. Furthermore, there is a correlation between the targets in Figure 18 that cause a fault in about one register on average and the targets in Figure 8 that generally result in persistent faults.

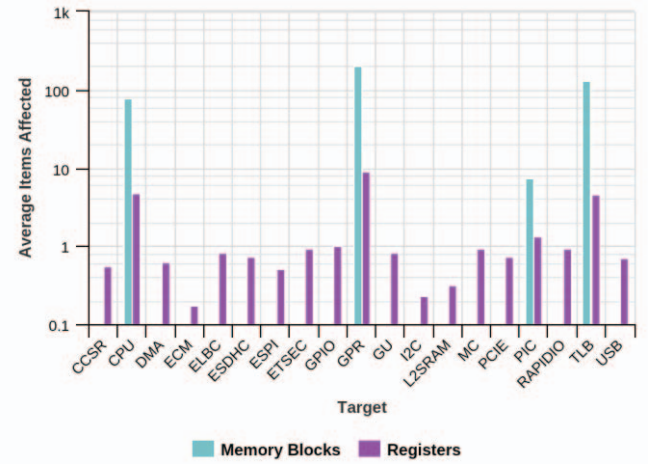

Figure 18. Fault propagation

7. Conclusion

This paper has presented our methodology for performing fault-injection testing on simulated commercial CPUs using Simics. The nature of the simulation allows us to observe, in great detail, the effects of injected faults on the system and target software. Our methodology allows designers to rapidly perform fault-injection campaigns to prepare for and supplement radiation-beam testing. We also presented a case study to demonstrate the powerful analysis made possible by our fault injector. We performed the case study's fault-injection campaign on a simulation of the Freescale P2020 running an image-processing application, which is representative of a real space-processing system.

Processor caches interesting for fault injection as caches occupy a large proportion of a device's area and therefore can account for a significant amount of SEUs within a device. While Simics does not include models of device caches by default, Simics does supply tools for creating cache models. We plan to augment the simulation of the P2020 with L1 and L2 caches so that we can target these caches for fault injection.

Work is currently underway to add support for the ARM Cortex-A9 processor to DrSEUs. We chose to add support for this device as there are two A9 cores included on Xilinx's Zynq SoC, which is featured in the CHREC Space Processor (CSP) [20] [21]. We are also working to expand our injection capabilities to physical devices by injecting faults via JTAG (for both Freescale P2020 and ARM A9), which will allow us to make comparisons between the results seen in simulation and those seen on physical devices.

Acknowledgments

This work was supported in part by the I/UCRC Program of the National Science Foundation under Grant No. EEC-0642422. The authors gratefully acknowledge the Simics toolset provided by Wind River that helped make this work possible.

References

[1] J. Engblom and D. Ekblom, "Simics: A Commercially Proven Full-System Simulation Framework," in *Workshop on Simulation in European Space Programmes*, 2006.

[2] P. Magnusson, M. Christensson, E. Jesper, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt and B. Werner, "Simics: A Full System Simulation Platform," *Computer,* vol. 35, no. 2, pp. 50-58, 2002.

[3] S. Guertin and M. Amrbar, "SEE Test Results for P2020 and P5020 Freescale Processors," in *Radiation Effects Data Workshop*, Paris, 2014.

[4] Space Micro, "Proton400k™ Single Board Computer," 9 May 2014. [Online]. Available: http://www.spacemicro.com/assets/datasheets/digital/slices/proton400k.pdf. [Accessed 18 October 2015].

[5] T. Karnik, P. Hazucha and J. Patel, "Characterization of Soft Errors Caused by Single Event Upsets in CMOS Processes," *IEEE Transactions on Dependable and Secure Computing,* vol. 1, no. 2, pp. 128-143, 2004.

[6] H. Quinn, D. Black, W. Robinson and S. Buchner, "Fault Simulation and Emulation Tools to Augment Radiation-Hardness Assurance Testing," *IEEE Transactions on Nuclear Science,* vol. 60, no. 3, pp. 2119-2142, 2013.

[7] H. Ziade, R. Ayoubi and R. Velazco, "A Survey on Fault Injection Techniques," *The International Arab Journal of Information Technology,* vol. 1, no. 2, pp. 171-186, 2004.

[8] M. Hsueh, T. Tsai and R. Iyer, "Fault Injection Techniques and Tools," *Computer,* vol. 30, no. 4, pp. 75-82, 1997.

[9] N. Wulf, G. Cieslewski, A. Gordon-Ross and A. George, "SCIPS: An Emulation Methodology for Fault Injection in Processor Caches," in *IEEE Aerospace*, 2011.

[10] R. Some, W. Kim, G. Khanoyan, L. Callum, A. Agrawal and J. Beahan, "A Software-Implemented Fault Injection Methodology for Design and Validation of System Fault Tolerance," in *Internation Conference on Dependable Systems and Networks*, Goteberg, 2001.

[11] R. Velazco, S. Rezgui and R. Ecoffet, "Predicting

Error Rate for Microprocessor-Based Digital Architectures through C.E.U. (Code Emulating Upsets) Injection," *IEEE Transactions on Nuclear Science,* vol. 47, no. 6, pp. 2405-2411, 2000.

[12] A. Benso, P. Prinetto, M. Rebaudengo and M. Reorda, "EXFI: A Low-Cost Fault Injection System for Embedded Microprocessor-Based Boards," *ACM Transactions on Design Automation of Electronic Systems,* vol. 3, no. 4, pp. 626-634, 1998.

[13] J. Carreira, H. Madeira and J. Silva, "Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers," *IEEE Transactions on Software Engineering,* vol. 24, no. 2, pp. 125-136, 1998.

[14] G. Cardarilli, F. Kaddour, A. Leandri, M. Ottavi, S. Pontarelli and R. Velazco, "Bit flip injection in processor-based architectures: a case study," in *IEEE Interational On-Line Testing Workshop*, 2002.

[15] R. Velazco, A. Corominas and P. Ferreyra, "Injecting Bit Flip Faults by Means of a Purely Software Approach: a Case Studied," in *IEEE Internation Symposium on Defect and Fault Tolerance in VLSI Systems*, 2002.

[16] B. Bastien, "A Technique for Performing Fault Injection in System Level Simulations for Dependability Assessment," Master Thesis, Univeristy of Virginia, 2004.

[17] W. Chao, F. Zhongchuan, C. Hongsong and C. Gang, "FSFI: A Full System Simulator-Based Fault Injection Tool," in *Internation Conference on Instrumentation, Measurement, Computer Communication and Control*, Beijing, 2011.

[18] D. Nussbaum, A. Fedorova and C. Small, "An overview of the Sam CMT simulator kit," Sun Microsystems, Inc., Mountain View, 2004.

[19] H. Guzman-Miranda, M. Aguirre and J. Tombs, "Noninvasive Fault Classification, Robustness and Recovery Time Measurement in Microprocessor-Type Architectures Subjected to Radiation-Inuduced Errors," *IEEE Transactions on Instrumentation and Measurement,* vol. 58, no. 5, pp. 1514-1524, 2009.

[20] C. Wilson, J. Stewart, P. Gauvin, J. MacKinnon, J. Coole, J. Urriste, A. George, G. Crum, E. Timmons, J. Beck, T. Flatley, M. Wirthlin, A. Wilson and A. Stoddard, "CSP Hybrid Space Computing for STP-H5/ISEM on ISS," in *Small Satellite Conference*, Logan, 2015.

[21] Space Micro, "Radiation Tolerant CHREC Space Processor," 23 April 2015. [Online]. Available: http://www.spacemicro.com/assets/datasheets/digital/slices/CHREC.pdf. [Accessed 18 October 2015].

BIOGRAPHY

Edward Carlisle *is a doctoral student in ECE at the University of Florida. He received B.S. degrees in EE and CEN and a M.S. in ECE from the University of Florida. He is a research assistant at the NSF Center for High-Performance Reconfigurable Computing (CHREC) studying fault-injection and mitigation for space applications.*

Nicholas Wulf *is a doctoral candidate in ECE at the University of Florida. He is a research assistant in the advanced processing devices group in the NSF CHREC Center at Florida. His research interests include analysis and comparison of fixed and reconfigurable device architectures and low-overhead fault-tolerant techniques.*

James MacKinnon *received his B.S. in Electrical Engineering from the University of Florida in 2014 and is currently pursuing an M.S with an expectation to graduate in 2016. He is a Research Assistant in the NSF CHREC Center in the field of hybrid space computing. His research interests include fault-tolerant space-computing systems and image processing.*

Alan D. George *is Professor of ECE at the University of Florida, where he serves as Director of the NSF Center for High-performance Reconfigurable Computing known as CHREC. He received the B.S. degree in CS and M.S. in ECE from the University of Central Florida, and the Ph.D. in CS from the Florida State University. Dr. George's research interests focus upon high-performance architectures, networks, systems, services, and applications for reconfigurable, parallel, distributed, and fault-tolerant computing. He is a Fellow of the IEEE.*