SCIPS: An Emulation Methodology for Fault Injection in Processor Caches

Nicholas Wulf, Grzegorz Cieslewski, Ann Gordon-Ross, Alan D. George
NSF Center for High-Performance Reconfigurable Computing (CHREC)
Department of Electrical and Computer Engineering, University of Florida
{wulf, cieslewski, ann, george }@chrec.org

Abstract—Due to the high level of radiation endured by space systems, fault-tolerant verification is a critical design step for these systems. [12]Space-system designers use fault-injection tools to introduce system faults and observe the system's response to these faults. Since a processor's cache accounts for a large percentage of total chip area and is thus more likely to be affected by radiation, the cache represents a key system component for fault-tolerant verification. Unfortunately, processor architectures limit cache accessibility, making direct fault injection into cache blocks impossible. Therefore, cache faults can be emulated by injecting faults into data accessed by load instructions. In this paper, we introduce SPFI-TILE, a software-based fault-injection tool for many-core devices. SPFI-TILE emulates cache fault injections by randomly injecting faults into load instructions. In order to provide unbiased fault injections, we present the cache fault-injection methodology SCIPS (Smooth Cache Injection Per Skipping). Results from MATLAB simulation and integration with SPFI-TILE reveal that SCIPS successfully distributes fault-injection probabilities across load instructions, providing an unbiased evaluation and thus more accurate verification of fault tolerance in cache memories.

TABLE OF CONTENTS

1. INTRODUCTION

Due to the vast amount of data produced by modern sensors and limited communication bandwidth, onboard data processing capabilities are an important concern for space-system designers. In addition, since some space systems operate at great distances from Earth, the communication delay necessitates a significant level of autonomy and real-time requirements for decision-making. For example, Mars landing vehicles may perform real-time terrain analysis during landing descent in order to locate the most appropriate landing site [1]. With such high computational demands, high-performance, many-core systems are becoming more attractive due to their balance of high processing capability and low power consumption compared to systems with one to several cores. However, high-performance space systems present several design challenges.

One of the primary challenges for high-performance space systems is fault-tolerant operation, which consists of accounting for and guarding against the high levels of space-born radiation. The radiation energy deposited in processing devices may cause device hardware faults in memory or control circuitry, possibly propagating into errors or even system crashes if unmitigated. Many fault-tolerant techniques exist to counter radiation effects, ranging from hardware-based fault masking to keep faults from manifesting into errors to software-based error detection and handling. Therefore, fault-tolerant verification is tantamount to the system's functionality design itself. A theoretically functionally "near-perfect" system may not be considered reliable until the system has undergone thorough fault-tolerant verification. However, fault-tolerant verification may be tedious, time-consuming, or unfeasible for certain fault types.

Automated fault-injection tools are highly effective for validating system design characteristics and demonstrating system robustness in the presence of certain faults. Fault-injection tools come in many forms with different testing abilities ranging from physical device irradiation using high-energy particles to running a software emulation of a faulty device. Typical fault-injection tools are designed to focus upon particular system components that are suspected to be particularly vulnerable to faults, such as cache and main memory, which constitute a majority of total chip area and thus have a larger probability of intercepting harmful radiation particles.

[1] 978-1-4244-7351-9/11/$26.00 ©2011 IEEE
[2] IEEEAC paper#1561, Version 2, Updated Jan 11, 2011

In order to assist system designers in fault-tolerant verification, we developed the Simple Portable Fault Injector (SPFI, pronounced "spiffy") tool in the NSF Center for High-Performance Reconfigurable Computing (CHREC) Center at the University of Florida. SPFI emulates device hardware faults using a debugger to alter values stored in main memory or registers during runtime. SPFI-TILE, targeted for Tilera's TILE64 device and its associated radiation-hardened version known as MAESTRO developed under the OPERA program at the National Reconnaissance Office (NRO), extends SPFI's functionality to support many-core device fault-tolerant verification. Although SPFI-TILE has access to certain device components such as the register file and main memory, similar to other software-based fault-injection tools, architectural limitations of the target device (TILE64 in this case) restrict SPFI-TILE from direct fault injection into caches.

In order to extend SPFI-TILE's fault-injection capabilities to effectively emulate cache faults without direct cache access, in this paper we propose a cache fault-injection methodology that emulates cache faults. Cache faults are emulated by pausing execution at a fault-injection point (at any instruction), identifying the next load instruction, and injecting a fault into the load instruction's accessed cache location prior to cache access. This cache fault-injection methodology leverages the fact that for a cache fault to manifest into a system error, a load instruction must access the faulty data. However, random fault-injection point selection results in the probability of a load instruction being chosen for fault injection being proportional to the number of non-load instructions preceding the load instruction (i.e., load instructions with higher numbers of preceding non-load instructions have a higher probability for fault injection). Thus, in order to provide thorough fault-tolerant verification, instruction-stream monitoring must mitigate this unbalanced cache fault-injection problem and distribute the injection probabilities evenly across all load instructions.

In this paper, we will present Smooth Cache Injection Per Skipping (SCIPS), a novel methodology to address the unbalanced cache fault-injection problem. SCIPS randomly *skips* load instructions to smooth out (i.e., balance) the injection probability across all load instructions. We show mathematically that this skipping process effectively performs a convolution on the load-instruction injection probabilities, and selecting a probability mass function (PMF) for determining the number of skips can be generalized to the Fourier analysis problem of filtering out all but the direct current (DC) bias. Results demonstrate that a relatively low number of average skips can dramatically even out the injection distribution, supporting the use of SCIPS as an effective fault-injection methodology for cache.

2. BACKGROUND AND RELATED WORK

Many space-system designers employ fault-injection tools for fault-tolerant verification. Fault-injection tools typically fall into one of three categories: hardware-, simulation-, and software-based.

Hardware-based, fault-injection methods inject faults into the physical device during normal operation by either direct manipulation of the device's pins or bombarding the device with radiation, such as with MESSALINE [3] and Gunneflo [4], respectively. The main advantage for hardware-based fault injection is that this method does not rely upon system models and/or assumptions, which may be flawed due to the difficulty of accurately modeling these systems. Since the system's physical hardware and software are tested, hardware-based injection tests for all system faults, even those not considered by the system designer. Furthermore, hardware-based injection is sometimes the only way to test certain faults, especially low-level VLSI circuitry faults. Unfortunately, hardware-based injection is sometimes not practical due to difficult or expensive testing devices such as heavy ion radiation fault-injectors. Additionally, hardware-based injection may even permanently damage devices, since the experiments often involve stimuli outside of the normal device specifications.

Unlike hardware-based injection, simulation-based injection does not require special hardware and alternatively injects faults into system models to predict the system's reaction to faults. System models can be designed to support a wide range of system abstraction levels. For instance, a low-level system model produces cycle-accurate results and a high-level system model abstracts away low-level internal workings in favor of simplicity and simulation speed. Examples of simulation-based injectors are CECIUM [5], which simulates a distributed application without the use of the actual source code and MEFISTO [6], which injects into VHDL-based model simulations [10]. Simulation-based injection is particularly useful in the early stages of system design when the full system may not exist yet. Since users work with simulations, users have full control over the faults, can target these faults to certain components or regions of code, and have an unobstructed view of all effects resulting from these faults. However, simulation-based injection has several drawbacks. High-level system models may exclude many of the design faults, resulting in inaccurate results. While low-level models can provide more accurate results, these models may suffer from lengthy simulation times and may also exclude some design faults. Furthermore, significant design time may be required to develop a simulator if no appropriate simulator exists.

Software-based injection provides a balance between hardware- and simulation-based methods, combining several advantages from both methods. As with hardware-based injection, the physical device is used to run the actual system software. However, instead of physically inserting faults into the device, which may do irreparable damage, the

system's state is modified during execution using software-based techniques such as manipulating the program state using a debugger or modifying the software to include routines for injecting logical errors and corrupting program variables. Similar to hardware-based injection, using the physical device reduces the time needed for running experiments and also automatically accounts for many hardware and software design faults. Similar to simulation-based injection, special and expensive hardware is not required and specific components and applications can be targeted on their own. Several existing software-based injectors include: DOCTOR [7] for distributed applications; NFTAPE [8], which provides a general framework for injecting into a wide variety of systems; and SPFFI [9], which injects into the configuration bits of a field-programmable gate array (FPGA) [10].

However, software-based injection suffers from two disadvantages. The first disadvantage is that software-based injection is highly intrusive, requiring the system to alter normal operation and perform self-injection (i.e., the system is responsible for fault injection into itself). For example, attaching a debugger to a process or inserting extra fault-injection routines into the code may alter the system's behavior. Unfortunately, these alterations may either mask or introduce new faults. The second disadvantage is that even though the software-based injection may have access to the memory and register file components, some highly vulnerable lower-level components, such as the cache, are hidden from the software. In this paper, we address this cache accessibility limitation and present a method to effectively *emulate* access to the cache via load instructions.

3. SPFI

SPFI represents a group of several software-based fault injectors (such as SPFI-µP and SPFI-TILE) that each target a specific device and aid system designers in fault-tolerant verification. In Section 3-1, we introduce SPFI-µP, which targets general single-core devices and is an appropriate example of the general SPFI method. In Section 3-2, we introduce SPFI-TILE, which targets the TILE64 using the same techniques as SPFI-µP.

3-1. SPFI-µP Method and Use

SPFI-µP emulates single-bit device hardware faults in microprocessors by inverting a single bit in main memory or the register file during runtime of a test program and observing any changed behavior from the test program. A SPFI-µP campaign is an automated series of such single-bit fault tests. Campaign parameters are set by the designer and include the location of the single-bit injections (e.g., a range of registers or memory locations) and how many tests should be run (typically hundreds or thousands).

Figure 1 shows the flow chart for a single SPFI-µP injection campaign. First, the designer specifies a code region for fault injection and sets campaign parameters. SPFI executes the program with a debugger attached, pauses at a randomly chosen fault-injection point in the tested code region, injects a fault, resumes program execution, and evaluates the program's results using a validation program. The validation program is designer-supplied and compares output results from SPFI-µP's execution of the test program with the actual expected correct results of the test program. This comparison reveals errors such as early program termination or missing data in an output file. SPFI repeats this process if more testing is required (specified by the designer in the campaign parameters) and outputs the final results summary, which includes both information on faults that caused errors and the specifics of each error, so as to allow the designer to diagnose system vulnerabilities.

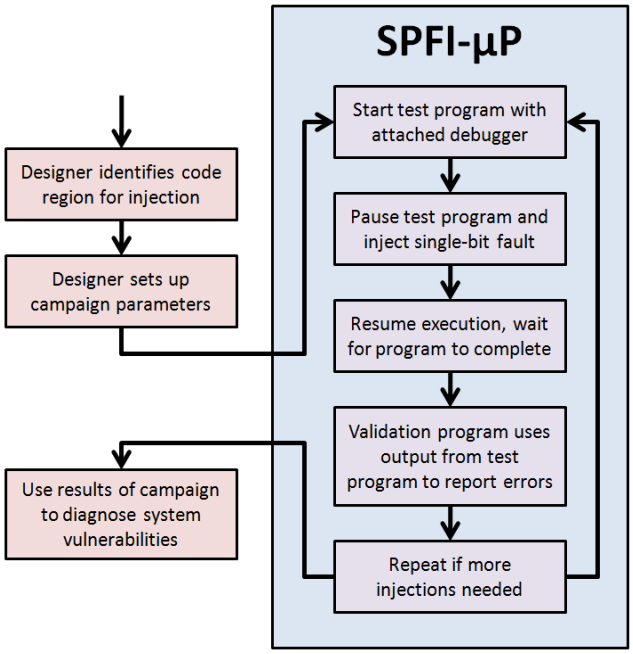


Figure 1: Flowchart for a single SPFI-µP injection campaign

SPFI-µP uses the debugger to pause/resume execution at the fault-injection points and read/write from/to main memory and the register file, effectively injecting faults into either component. In addition to main memory and register file fault injection, SPFI-µP emulates instruction-cache faults by reading the program counter and modifying the next instruction before the next instruction is executed. Since the instruction cache never flushes back to main memory (without loss of generality, we assume no self-modifying code), induced errors in instruction cache are nearly always transient. To effectively emulate transient errors, SPFI-µP provides a campaign parameter to allow the designer to specify a *transient number*, which determines the number of instructions SPFI-TILE should step through after an injection before correcting the fault (i.e., resetting the corrupted bit back to the original value stored immediately before injection).

Due to architectural limitations of devices and of software-based injectors in general, SPFI-μP cannot provide information on how likely a fault is to occur in certain areas of the device. In addition, SPFI-μP cannot predict how certain hardware-based, radiation-hardening techniques may perform, such as Single Error Correcting Double Error Detecting (SECDED) caches or Dual Interlocked Cell (DICE) flip-flops used in radiation-hardened devices. Such information and analysis is beyond the scope of SPFI-μP and must be collected by the designer, likely through physical radiation testing.

However, SPFI-μP does provide a description of the likely effects that a single-bit error will produce. Although certain devices may be hardened against radiation, there is still a very good chance that several errors will occur within the lifetime of the system. SPFI-μP is useful in identifying high-vulnerability components as well as testing the effectiveness of software-based, fault-tolerant techniques. Since the designer selects specific code regions for fault injection, SPFI-μP can also be used to expose the vulnerabilities within a program to help guide the focus of any fault-tolerant approaches. For example, SPFI-μP results targeted at various sections of code may reveal that a single section is far more vulnerable to faults than any other. Rather than wasting resources on the entire program, the designer can focus their efforts on adding fault tolerance to the single vulnerable section. Moreover, with SPFI-μP's ability to perform injection campaigns consisting of thousands of single-bit injection tests, SPFI-μP can quickly test many potential designs without the need for lengthy, costly forms of physical radiation testing.

3-2. Extension to Many-Core

In order to address the increasing interest in using many-core devices in hazardous environments, SPFI-TILE was developed to apply the SPFI-μP injection scheme to the many-core TILE64. Ignoring added functionality, the basic process of fault injection used in SPFI-TILE is identical to the SPFI-μP process. Furthermore, since the MAESTRO device was designed to mimic the TILE64 from a software perspective, SPFI-TILE can be used to inject into the MEASTRO with little alteration.

SPFI-TILE adds several functionalities as compared to SPFI-uP. Prior to beginning an injection campaign, a system designer may identify a subset of tiles to test within the set of 64 tiles (a tile refers to a single microprocessor core in the TILE64) using the campaign parameters. Since SPFI-TILE's purpose is to analyze the effects of single-bit upsets, SPFI-TILE only needs to inject a single-bit fault into a single tile for each individual test. To satisfy this single-bit injection model, a random tile is selected from the user-specified subset of tiles at the beginning of each test. Once program execution reaches the beginning of the user-selected code region to test, SPFI-TILE attaches a debugger to the selected tile and injects a fault according to the same process used by SPFI-uP in Figure 1.

4. Smooth Cache Injection Per Skipping (SCIPS)

In this section we describe SCIPS, a cache fault-injection extension to SPFI-TILE, which evenly distributes fault-injection probabilities across all load instructions (i.e., cache locations). In Section 4-1, we present the basic cache fault-injection methodology. In Section 4-2, we present motivation for SCIPS using an example to show how naïve, random, fault-injection point selection results in unbalanced cache fault injection. Sections 4-3 and Section 4-4 present SCIPS and a theoretical analysis, respectively.

4-1. Data-cache Fault-Injection Methodology

As with most debuggers, the well-known GDB tool provides no direct cache access due to a device's architectural limitations. Whereas SPFI-TILE already supports instruction-cache fault injections, data-cache fault injection is equally important for complete system fault-tolerance verification. Fortunately, similarly to instruction-cache fault injection, data-cache fault injection is possible to emulate using GDB's functionality.

In order for a cache fault to manifest into a system error, a load instruction must request the faulty cache data. SPFI-TILE can emulate cache faults by pausing execution at a fault injection point, stepping through instructions until the next load instruction, and then injecting an error into the appropriate memory location before the memory is accessed. Unfortunately, this naïve First Load Instruction Injection Methodology (FLIM) has some limitations and does not consider all execution scenarios. Firstly, SPFI-TILE does not know if the faulty cache data will be flushed back to main memory (i.e., if a transient fault becomes a permanent fault). To account for transient cache-data faults, a designer-set transient number specifies when faulty cache data should be corrected, which is akin to the transient number SPFI-TILE already uses for instruction-cache fault injection. Secondly, faulty cache data should only manifest into an error if the faulty cache data is either loaded into a register or flushed back to main memory before the faulty cache data is either written over by a subsequent store instruction or invalidated due to normal cache operations. Thirdly, due to hidden cache behaviors, certain load instructions may be more or less likely to access faulty cache data, resulting in unequal fault-inducing probabilities across load instructions. For example, load instructions with a high cache-miss frequency are less likely to access faulty cache data because the newly loaded data from main memory (which would rarely be faulty due to inherent error correction mechanisms present in main memory) would overwrite clean, but faulty, cache data. Therefore, these load instructions would have lower relative fault-inducing probabilities.

Unfortunately, these last two points are exceedingly difficult to resolve and are therefore beyond the scope of SPFI-TILE. Similarly to SPFI-TILE's instruction-cache fault injection,

SPFI-TILE's data-cache fault injection only considers faulty cache data that has already manifested into a device error.

Finally, as with most software-based, fault-injection tools, SPFI-TILE's system description is not guaranteed to be perfect, but the general benefits attained via fault-tolerance verification time and monetary cost often balance these shortcomings.

4-2. Drawbacks to the First Load Instruction Methodology (FLIM)

Since actual cache behavior is hidden from SPFI-TILE, SPFI-TILE cannot deduce which load instructions are more likely to access faulty cache data (see Section 4-1) and therefore must assume that all load instructions have an equal fault-inducing probability. Unfortunately, FLIM does not satisfy this equal fault-inducing probability and instead introduces unbalanced, cache fault injection by favoring load instructions that are preceded by a larger number of non-load instructions. We exemplify this unbalanced cache injection problem using the example given in Table 1, which depicts a simple five-instruction loop with two load instructions followed by three non-load instructions.

Table 1: Example of Unbalanced Injection Problem

Address	Instruction
1st	Load_1
2nd	Load_2
3rd	Non-Load
4th	Non-Load
5th	Jump to 1st Address

This assembly code results in the same five instructions being repeated indefinitely. Without affecting the contribution and functionality of this example, we can assume that all instructions require the same amount of time to execute.

SPFI-TILE can randomly choose any of these five instructions as a fault-injection point. If the fault-injection point is a load instruction, that instruction is selected for fault injection. If the fault-injection point is a non-load instruction, SPFI-TILE steps through the subsequent instructions until the next load instruction is encountered. For example, if SPFI-TILE selects the second address as the fault-injection point, Load_2 would be selected for fault injection. However, if SPFI-TILE selects any of the other four instructions, Load_1 would be selected for fault injection. Under the assumption that all instructions require equal execution time and have equal fault-inducing probabilities, SPFI selects Load_1 for fault injection 80% of the time. This unbalanced fault injection may result in biased vulnerability measurements for this code region.

Unbalanced fault injection in caches is unrelated to the unequal fault-inducing probability as a result of the hidden cache behavior. Unbalanced fault injection is simply an artifact of the instruction execution flow and in no way represents the actual fault-inducing probabilities that would be experienced in a faulty environment. For example, even if Load_1 had a high cache miss rate, Load_1 would still be responsible for 80% of the injected faults even though Load_1 actually has a low fault-inducing probability. Therefore, ensuring balanced fault-injection probabilities results in more accurate fault-tolerance verification.

4-3. Balancing Cache-Fault Injection Probabilities using SCIPS

We introduce SCIPS as a novel method for balancing cache fault-injection probabilities. Instead of naively injecting into the first load instruction after a fault-injection point, SCIPS randomly skips several load instructions. We exemplify SCIPS using the same five-instruction loop from Table 1.

SCIPS identifies fault-injection points using the method as described in Section 3. However, when SCIPS identifies the next load instruction for fault injection, there is a 50% probability that SCIPS skips the First Load Instruction (FLI) encountered, and instead selects the second load instruction after the fault-injection point for fault injection. We elaborate on our selection of a 50% skipping probability in Section 4-4.

Considering Load_1's 80% fault-injection probability from Section 4-3, with SCIPS, Load_1 still has an 80% probability of being the FLI, but when combined with the 50% skip probability, both Load_1 and Load_2 result in a 40% fault-injection probability when Load_1 is the FLI. Similarly, Load_2 has a 20% chance of being the FLI, but when combined with the 50% skip probability, both Load_1 and Load_2 result in a 10% fault-injection probability when Load_2 is the FLI. The addition of these fault-injection probabilities results in a 50% fault-injection probability for both load instructions, which provides a dramatic improvement over the initial 80% and 20% fault-injection probabilities from Section 4-3.

4-4. SCIPS Theoretical Analysis

Even though the five-instruction example represents a simplified code region, SCIPS remains effective for more complex code by increasing the skip range. For example, a realistic code region my require skip ranges as high as 30, meaning a random number between 0 and 30 would be chosen on each fault-injection test to determine the number of skips to be performed after finding the FLI. In this section, we generalize SCIPs and provide a theoretical analysis.

The effects of SCIPS are analogous to the effects in signal processing when applying a Finite Impulse Response (FIR) filter to a noisy signal. This relationship enables accurate

predictions of SCIPS' effects on a given program. We exemplify this relationship by stepping through a generalized SCIPS example.

Considering a general region of executing instructions, for each load instruction we record the number of non-load instructions preceding that load instruction in an array *Code*, where *Code*[*n*] equals 1 (counting the current load instruction) plus the number of non-load instructions preceding the n^{th} load instruction in this code region. Assuming arbitrary loop iterations, the code from Table 1 produces:

$$Code[n] = \{1,4,1,4,1,4, \dots\}$$

If $P_{FLIM}[n]$ equals the probability of the n^{th} load instruction being the FLI after the fault-injection point, $P_{FLIM}[n]$ is also the fault-injection probability of the n^{th} load instruction using FLIM. $P_{FLIM}[n]$ can be calculated by dividing *Code*[*n*] by the total number of executed instructions. Assuming a total of 100 executed instructions, $P_{FLIM}[n]$ becomes:

$$P_{FLIM}[n] = \left\{\frac{1}{100}, \frac{4}{100}, \frac{1}{100}, \frac{4}{100}, \frac{1}{100}, \frac{4}{100}, \dots\right\}$$

Let $P_{skip}[n]$ represent the Probability Mass Function (PMF) of skipping *n* load instructions before injecting into the $(n+1)^{th}$ load instruction. In practice, $P_{skip}[n]$ is defined by the designer to produce the desired results of balanced cache fault injection. Typical effective distributions have a skip range close to 20, with better results provided as the skip range increases. Defining $P_{skip}[n]$ is analogous to defining an effective moving average filter for a noisy signal. Using the example from Section 4-3 with a skip range of 1 (50% skip probability) produces:

$$P_{skip}[n] = \left\{\frac{1}{2}, \frac{1}{2}, 0,0,0,0, \dots\right\}$$

Finally, let $P_{SCIPS}[n]$ equal the fault-injection probability of injecting into the n^{th} load instruction using SCIPS with a skipping PMF equal to $P_{skip}[n]$.

Next, we consider the computation involved in calculating $P_{SCIPS}[n]$ for a given *n* using the following pattern: 1) calculate the probability of not skipping when the n^{th} load is the FLI; 2) calculate the probability of skipping once when the FLI is the $(n-1)^{th}$ load instruction; 3) calculate the probability of skipping twice when the FLI is the $(n-2)^{th}$ load instruction; 4) continue this pattern. The summation of all these conditional probabilities produces the final probability that the n^{th} load instruction is selected for fault injection.

The two probabilities of skipping *m* times and selecting a fault-injection point *m* load instructions before the n^{th} load instruction (represented by $P_{skip}[m]$ and $P_{FLIM}[n-m]$ respectively) are based on two independent events, meaning the probability of the intersection of these two events is equal to the product of $P_{skip}[m]$ and $P_{FLIM}[n-m]$. Therefore:

$$P_{SCIPS}[n] = \sum_{m=0} P_{FLIM}[n-m] \cdot P_{skip}[m]$$

Thus, $P_{SCIPS}[n]$ is equal to $P_{FLIM}[n]$ convolved with $P_{skip}[n]$.

$$P_{SCIPS}[n] = \left(P_{FLIM} * P_{skip}\right)[n]$$

Since the goal of SCIPS is to balance fault-injection probabilities, $P_{SCIPS}[n]$ should be flat. Therefore, the challenge in creating an appropriate $P_{skip}[n]$ for SCIPS is equivalent to creating a FIR filter to remove as much of the alternating current (AC) from the signal composed of the original uneven injection probabilities.

5. RESULTS AND ANALYSIS

We demonstrate the fault-injection balancing capabilities of SCIPS for cache using two experiments: a MATLAB simulation as a proof of concept design using randomly generated code; and integration of SCIPS into SPFI-TILE to evaluate effectiveness of SCIPS on real application code.

5-1. MATLAB SIMULATION

To demonstrate the ability of SCIPS to effectively perform convolution on the signal produced by the load distribution in the instruction flow, we generate synthetic code using MATLAB and use this synthetic code to simulate cache fault injections. A Poisson process distributes the loads throughout a randomly generated set of 200 instructions with an average of one load for five instructions (we determined this estimate to be realistic based on several actual code samples) and this set of 200 instructions loops 25 times. MATLAB then uses this synthetic code to produce a theoretical prediction of SCIPS' effects based on the signal analysis theory presented in Section 4-4. Finally, the synthetic code is used to simulate actual random fault-injection results by determining a fault-injection point (a pseudo-random number between one and the maximum number of instructions) and using SCIPS to determine the load instruction for fault injection. Our simulation runs 100,000 injections over an average of 1,000 load instructions for an average of 100 injections per load. SCIPS uses a skipping range of 20 (skips 0 to 20 times), thereby representing a moving average filter of width 21.

Figure 2 and Figure 3 show the theoretical prediction and simulated fault-injection probabilities, respectively, for both FLIM (highly peaked line) and SCIPS (smoother line) using the synthetic code. The figures show the unbalanced fault-injection probabilities using FLIM and the ability of SCIPS to significantly smooth the fault-injection probability. Furthermore, the close match between the theoretical results (Figure 2) and the simulation results (Figure 3) further supports our convolution analysis theory.

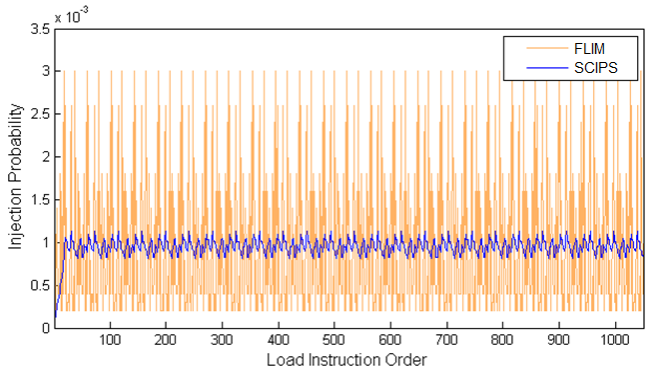


Figure 2: Theoretical prediction of fault-injection probabilities across all load instructions for synthetic code using FLIM (highly peaked line) and SCIPS (smoother line)

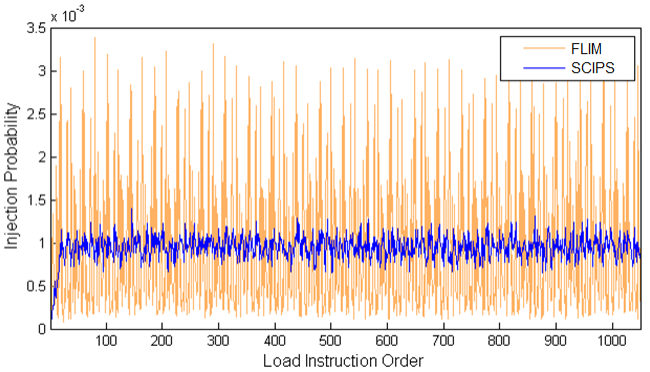


Figure 3: Simulated fault-injection probabilities across all load instructions for synthetic code using FLIM (highly peaked line) and SCIPS (smoother line)

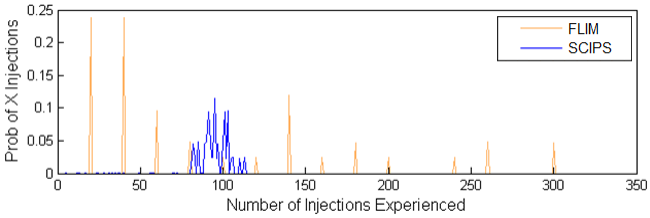


Figure 4: Theoretical injection count PFM for a given load instruction using FLIM (dispersed groups) and SCIPS (tight grouping)

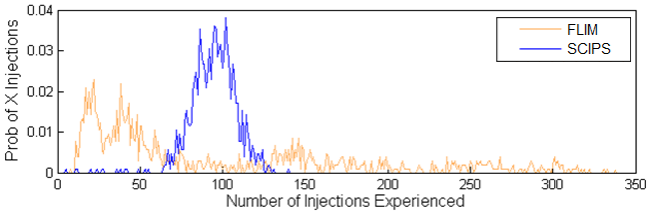


Figure 5: Simulated injection count PFM for a given load instruction using FLIM (wider grouping) and SCIPS (tight grouping)

Figure 4 and Figure 5 show the injection count PMFs for any given load instruction based on the theoretical prediction and simulated results, respectively, for the synthetic code. Figure 5 shows the peak value of 3.81% for SCIPS occurs at a value of 102 on the horizontal axis. This result means that 3.81% of the load-instruction instances in the synthetic code each accounted for 102 fault injections. The fact that the SCIPS values have a tighter grouping around 100 injections shows the effectiveness of SCIPS in balancing the fault injections. As Figure 5 illustrates, SCIPS ensures that 99% of all load instructions receive between 50 and 150 injections. Additionally, the variance of FLIM results is 0.6985 compared to 0.0203 for SCIPS, a 97% reduction.

5-2. SPFI-TILE Simulation

We integrated SCIPS into SPFI-TILE to verify the effectiveness of SCIPS with real application code. We used matrix multiply (a kernel algorithm common in many space-applications such as hyperspectral imaging [2]) with a given number of tiles used to calculate the product matrix *C* from the input matrices *A* and *B*. The matrix multiply algorithm distributed the workload by breaking the *C* matrix into blocks of rows, each of which was assigned to a single tile for computation. Figure 6 shows the code region responsible for calculating *C* from *A* and *B*'s transpose. This code region accounts for nearly all of the computation time, and is thus the focus of our fault-injection experiments. With *A* and *B* dimensions of 550×600 and 600×650, respectively, and five tiles for computation, execution required four seconds on the TILE64.

```
for (int i=0; i<rowCount; i++) {
   for (int j=0; j<dimWC; j++) {
      int sum = 0;
      for (int k=0; k<dimInner; k++) sum += matA[i][k] * matBtp[j][k];
      matC[i][j] = sum;
   }
}
```

Figure 6: Matrix multiply code used for SPFI-TILE SCIPS simulation

The computation in Figure 6 processes every location in the *C* matrix and performs the highlighted FOR loop to calculate that location's value. Since the inner dimension of *A* and *B* is 600, the highlighted FOR loop iterates 600 times before moving on to the next location in the *C* matrix. Therefore, over 99% of fault injections into this code region will result in a fault injection into the highlighted FOR loop. Therefore, the remainder of our analysis focuses only on the load instructions within the FOR loop.

Table 2 depicts the 40 addresses corresponding to the FOR loop's instructions. For compactness, we list the instruction's base address in multiples of eight (11D80 to 11EC0 in hexademical) in the leftmost column and the eight instructions offset from that base address in each row. Instructions outside of the FOR loop have been grayed out

and marked with a dash sign. The highlighted cells indicate the load instructions and the numbers within these cells show the probability of that load instruction being the FLI assuming all instructions have equal computation time. The load instructions at addresses 11DB0 and 11EC0 have 5 and 7 preceding non-load instructions, respectively, and thus have the highest fault-injection probabilities when using FLIM.

Table 2: Range of addresses used in FOR loop using 8-byte instruction words with gray dash cells used for instructions outside of FOR loop and red cells used for load instructions with FLI probabilities

Base Addr.	Offset							
	00	08	10	18	20	28	30	38
11D80	-	-	-	-	-		15%	
11DC0	5%				10%		5%	
11E00	5%		5%		5%		5%	
11E40	5%		5%		5%		5%	
11E80	5%							
11EC0	20%					-	-	-

Using SCIPS, SPFI-TILE ran fault-injection campaigns of 5,000 fault injections on the code region in Figure 6. Even though SPFI-TILE did not actually perform the fault injections (actual fault injection was not necessary to show effectiveness of SCIPS), SPFI-TILE reported the load instruction addresses that were selected for fault injection using SCIPS. Figure 7 shows the results of the SPFI-TILE experiment using different values for the skip range.

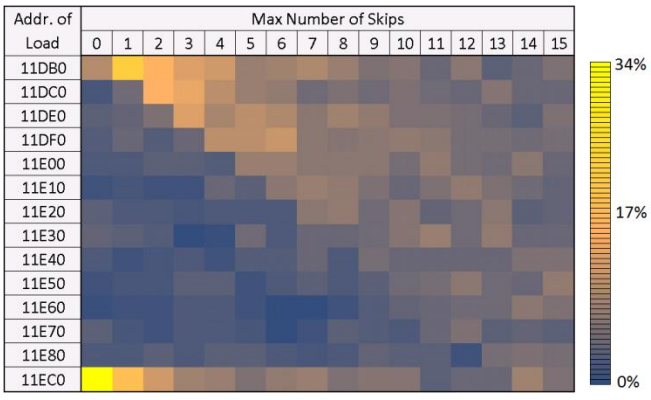


Figure 7: Distribution of injections for increasing skipping values

As predicted, load instructions at 11DB0 and 11EC0 receive the most injections using low skip counts. However, as skip counts increase, the distribution of injections quickly balances out. The variation within the rightmost columns does not necessarily indicate residual unevenness in the probabilities. Even if all load instructions were perfectly equiprobable, some variation in actual injections experienced would still occur due to the variation in the randomly generated variable used for the number of skips.

6. CONCLUSIONS

In this paper, we have introduced SCIPS (Smooth Cache Injection Per Skipping), a novel methodology to ensure even probabilities for fault injection in caches across all load instructions. In addition to a mathematical analysis of SCIPS, results of simulating SCIPS in MATLAB using synthetic code support the theory behind SCIPS, confirming that SCIPS resembles a FIR filter in the way it operates on and evenly distributes load-instruction, fault-injection probabilities, reducing the variance of the probability distribution by 97%. SCIPS is then integrated with the SPFI-TILE fault-injection tool, and a case study is performed using a matrix-multiply application, the results of which further demonstrate effectiveness of SCIPS. Although the naïve approach (FLIM) produces fault-injection probabilities as high as 33% for certain load instructions, SCIPS brings all probabilities within the range of 6% - 9% with skipping ranges as small as 15 skips.

The main contribution provided by SCIPS is that it enables both researchers in fault-tolerant computing and space-system designers to more effectively test a device's cache while still benefiting from a software-based, fault-injection tool. In order to verify the accuracy of SCIPS as compared to direct cache access, future work includes comparing SCIPS to simulation-based, fault-injection methodologies, such as those using SimpleScalar, which provide direct cache access.

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 vendor equipment and tools provided by the National Reconnaissance Office that helped make this work possible.

REFERENCES

[1] S. Ploen, C. Kinney, and H. Seraji, "Determination of Terminal Landing Footprint for On-Board Terrain Assessment and Intelligent Hazard Avoidance," *2003 AIAA Guidance, Navigation, and Control Conference and Exhibit*, Aug 11-14, 2003.

[2] Landgrebe, D.; , "Hyperspectral image data analysis," *Signal Processing Magazine, IEEE*, vol.19, no.1, pp.17-28, Jan 2002

[3] Arlat, J.; Aguera, M.; Amat, L.; Crouzet, Y.; Fabre, J.-C.; Laprie, J.-C.; Martins, E.; Powell, D.; , "Fault injection for dependability validation: a methodology

and some applications," *Software Engineering, IEEE Transactions on* , vol.16, no.2, pp.166-182, Feb 1990.

[4] Gunneflo, U.; Karlsson, J.; Torin, J.; , "Evaluation of error detection schemes using fault injection by heavy-ion radiation," *Fault-Tolerant Computing, 1989. FTCS-19. Digest of Papers., Nineteenth International Symposium on* , vol., no., pp.340-347, 21-23 Jun 1989.

[5] G. Avarez and F.Christian , "Centralized failure for distributed, fault-tolerant protocol testing," in *Proceedings of the 17th IEEE International Conference on Distributed Computing Systems (ICDCS'97)*, May 1997.

[6] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, and J. Karlsson, "Fault Injection into VHDL Models: The MEFISTO Tool", *Proceedings of the 24th International Symposium on Fault-Tolerant Computing*, June 15-17, 1994.

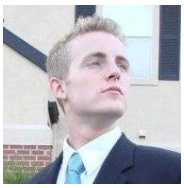
[7] S. Han, K. Shin, and H. Rosenberg. "Doctor: An integrated software fault injection environment for distributed real-time systems", *Proc. Computer Performance and Dependability Symposium, Erlangen*, Germany, 1995.

[8] D.T. Stott and al. Nftape: a framework for assessing dependability in distributed systems with lightweight fault injectors. In *Proceedings of the IEEE International Computer Performance and Dependability Symposium*, pages 91-100, March 2000.

[9] G. Cieslewski and A. George, "SPFFI: Simple Portable FPGA Fault Injector," *Proc. of Military and Aerospace Programmable Logic Devices Conference (MAPLD)*, Greenbelt, MD, Aug. 31 - Sep. 3, 2009.

[10] S. Tixeuil,W. Hoarau, and L. Silva. An overview of existing tools for fault-injection and dependability benchmarking in grids. In *Second CoreGRID Workshop on Grid and Peer to Peer Systems Architecture*, Paris, France, January 2006.

BIOGRAPHY



Nicholas Wulf *is a Ph.D. student in Electrical and Computer Engineering at the University of Florida. He is a research assistant in the Advanced Space Computing group at the High-Performance Computing and Simulation Research Laboratory. His research interests include fault-tolerant architectures and techniques for FPGA and many-core based systems.*

Grzegorz Cieslewski *is a graduate student at the University of Florida where he is currently pursuing a Ph.D. degree in Electrical and Computer Engineering. As a research assistant he is a member of the Advanced Space Computing and Reconfigurable Computing groups at the High-performance Computing & Simulation Research Laboratory. His research interests include computer architecture, reconfigurable, fault-tolerant and distributed-computing as applied to linear algebra and signal processing problems. He is a student member of IEEE.*

A. Gordon-Ross *(M'00) received her B.S and Ph.D. degrees in Computer Science and Engineering from the University of California, Riverside (USA) in 2000 and 2007, respectively. She is currently an Assistant Professor of Electrical and Computer Engineering at the University of Florida (USA) and is a member of the NSF Center for High Performance Reconfigurable Computing (CHREC) at the University of Florida. She is also the faculty advisor for the Women in Electrical and Computer Engineering (WECE) and the Phi Sigma Rho National Society for Women in Engineering and Engineering Technology. She received her CAREER award from the National Science Foundation in 2010 and Best Paper awards at the Great Lakes Symposium on VLSI (GLSVLSI) in 2010 and the IARIA International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies (UBICOMM) in 2010. Her research interests include embedded systems, computer architecture, low-power design, reconfigurable computing, dynamic optimizations, hardware design, real-time systems, and multi-core platforms.*

Alan D. George *is Professor of Electrical and Computer Engineering at the University of Florida, where he serves as Director of the HCS Research Lab and Director of the new NSF Center for High-performance Reconfigurable Computing (CHREC). He received the B.S. degree in Computer Science and the M.S. in Electrical and Computer Engineering from the University of Central Florida, and the Ph.D. in Computer Science from the Florida State University. Dr. George's research interests focus upon high-performance architectures, networks, services, and systems for parallel, reconfigurable, distributed, and fault-tolerant computing. He is a senior member of IEEE and SCS, a member of ACM and AIAA, and can be reached by e-mail at ageorge@ufl.edu.*