SAND2021-4082J

Multi-scale System Modeling of Single Event Induced Faults in Advanced Node Processors

Matthew Cannon, Arun Rodrigues, Dolores Black, Jeff Black, Luis Bustamante, Matthew Breeding, Ben Feinberg, Micahel Skoufis, Heather Quinn, Lawrence T. Clark, John Brunhaver, Hugh Barnaby, Michael McLain, Sapan Agarwal and Matthew J. Marinella

***Abstract*—Integration-technology feature shrink increases computing-system susceptibility to Single-Event Effects (SEE). While modeling SEE faults will be critical, an integrated processor's scope makes physically correct modeling computationally intractable. Without useful models, pre-silicon evaluation of fault-tolerance approaches becomes impossible. To incorporate accurate transistor-level effects at a system scope, we present a multi-scale simulation framework. Charge collection at the (i) device-level determines (ii) circuit-level transient duration and state-upset likelihood. Circuit effects, in turn, impact (iii) register-transfer-level architecture-state corruption visible at the (iv) system-level. Thus, the physically accurate effects of SEEs in large-scale systems, executed on an HPC simulator, could be used to drive cross-layer radiation hardening by design. We demonstrate the capabilities of this model with two case studies. First, we determine a D flip-flop's sensitivity at the transistor level on 14nm FinFet technology, validating the model against published cross-sections. Second, we track and estimate faults in a MIPS processor for Adams 90% worst-case environment in an isotropic space environment**

***Index Terms*—Single event effects (SEE), single event transient (SET), single event upset (SEU), fault modeling, structural simulation toolkit (SST)**

I. Introduction

SINGLE-event effects (SEEs) are a key radiation susceptibility for electronic components fabricated with modern highly scaled process technologies [1]. For microprocessors, understanding the effect of SEEs on the architecture is challenging, due to the number of SEE-sensitive locations and the inability to observe the architecture fully. From previous work, we understand that microprocessors are susceptible to silent data corruption (SDC), crashes, and halts from single-event upsets (SEUs) and single-event transients (SETs) in the control and data flow of the microprocessor architecture [2].

M. Cannon (mcannon@sandia.gov), A. Rodrigues, D. Black, J. Black, B. Feinberg, M. Breeding M. McLain and M. Marinella (mmarine@sandia.gov) are with Sandia National Laboratories, Albuquerque, NM.

L. Bustamante, M. Skoufis and S. Agarwal (sagarwa@sandia.gov) are with Sandia National Laboratories, Livermore, CA.

H. Quinn is with Los Alamos National Laboratories, Los Alamos, NM.

L. Clark, J. Brunhaver and H. Barnaby are with Arizona State University, Tempe AZ.

The authors acknowledge the use of Monte Carlo Radiative Energy Deposition (MRED) software under the license from Vanderbilt University.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

System designers need to understand how errors flow through microprocessor-based systems to understand how faults in the microprocessor affect the overall system. In many ways, while halts and crashes are the hardest issue to mitigate for system design, handling SDC faults is likely the more challenging problem because the undetected error can affect several components. Most SDC faults are written back into dynamic random access memory (DRAM) for use in other calculations, which means that the fault transitions from one calculation to DRAM then back to the microprocessor in other calculations. As systems scale into larger installations, it might also be necessary to understand the effect of SEEs in multiple-microprocessors systems, which makes the scalability of tracking errors more challenging. Finally, accounting for varying radiation sensitivity, component architectures, and functionality in large-scale systems requires an understanding of different component architectures and their radiation sensitivities. Therefore, fault modeling for large-scale system integration is necessary, but does not exist currently.

A holistic approach to model the effects of these faults is needed. The current standard, radiation testing, does not provide enough insight to how a microprocessor fails. Developing an understanding of microprocessors from radiation testing can be a complex and time consuming process. Fault simulation and emulation techniques can be useful at providing an initial understanding of how the microprocessor architecture and software respond to faults that can be verified in radiation testing.

Fault simulation tools are advantageous over many modeling options, since they expose the underlying architecture. Some of the most complicated design in modern electronics is the intricate timing and data coordination in the pipeline stages of a microprocessor (instruction fetching, ALU operations, memory loading, storing data, etc, as shown in Fig. 1). Tools that understand the microprocessor architecture well enough to translate from faults in the transistor level to errors in the pipeline structure are necessary to fully understand the robustness of the given architecture.

To address this need, a multi-scale simulation framework to model the effects of SEUs and SETs has been developed. The framework considers several levels of abstraction to model SEEs in microprocessors at both the software and transistor level. The framework is illustrated in Fig. 2. At the software layer, an algorithm in C or C++, is compiled to individual assembly instructions that can be run on a discrete event simulation (DES) model of a faulty processor. SEUs are injected

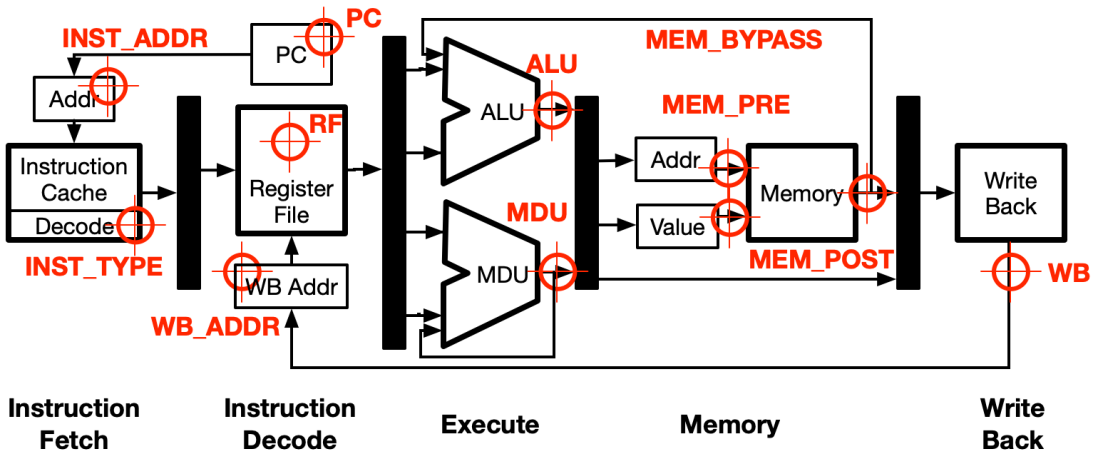


Fig. 1: The layout of the five-stage pipeline with locations where faults are inserted. The black bars show the location of pipeline registers that break the computation into distinct and separate stages.

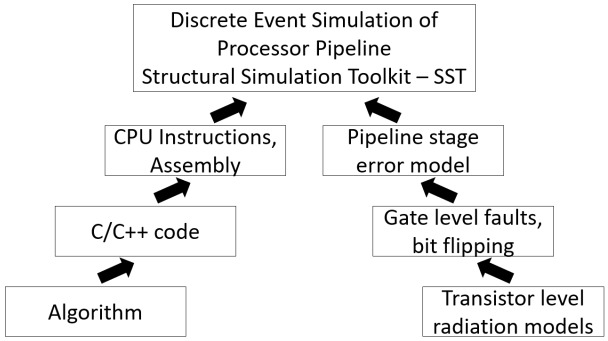


Fig. 2: The impact of transistor level radiation effects on a running algorithm are modeled through a discrete event simulation of the microprocessor pipeline.

into the registers of a simulated system. The probability of an SEU on each register is calculated using transistor level SEE sensitivities. As the effective cross-section is dominated by the registers themselves and not the combinational logic, we approximate the cross-section as the register cross-section.

The paper starts with a discussion of related work on fault injection techniques in Section II. The fault injection methodology in this paper is described in detail in Section III. The paper includes two case studies in Section IV. The first case study is the simulation of a D Flip-Flop. The second case study is a full microprocessor implementation with different software modifications. We end the paper with conclusions in Section V.

II. Related Work

Fault injection is the generic term for all methods of injecting faults into a system, including both fault emulation and fault simulation [3]. Fault emulation encompasses fault injection methods that insert faults directly into the hardware platform. Fault simulation methods focus on injecting faults into a software simulation of the hardware architecture. In this section, other methods of fault emulation and simulation previously used on microprocessor architectures are covered.

Fault emulation on microprocessors is common, because it does not require complete understanding of the underlying architecture, which is often proprietary. The most common techniques for microprocessor fault emulation rely on interrupts, compilers, boundary scan, or software fault injection to insert the faults into the component. The most common method is software-based fault emulation that uses software modification to insert SEUs into the microprocessor [4]–[7]. Inserting faults into software through the compiler is uniquely common to the open-source LLVM compiler, which means that several LLVM-based fault insertion tools have been developed, including LLFI, KULFI, ClAP (Critical Instruction Analysis and Protection), and Relax [8]–[11]. Compiler-based modification of the software is attractive, as the compiler can handle the software modifications and is able to focus specifically on the compiler's intermediate representation or the machine instructions, instead of software expressions. Interrupt-based methodologies attempt to inject faults into registers or the cache using interrupt controller routines [12], [13]. Boundary scan ports, including the Joint Test Action Group (JTAG) interface, can also insert faults directly into microprocessors without software modification [14]–[16]. Inserting faults through the debugger is also possible automatically or by hand [17]. While fault emulation techniques are often faster than fault simulation, in this case the hindrance of not being able to inject faults across the entire architecture is a problem. Furthermore, the instrumentation of the software or interrupt controllers limits the scope of the injection sites to registers. As shown in [18], [19], these software-based techniques can be faulty and vary by as much as 45 times based on the method used. Therefore, fault emulation on microprocessors can be limited.

Fault simulation can be useful to microprocessors, as long as a complete understanding of the architecture is possible. As information scarcity is common in microprocessor design, the ability to get a complete understanding of the architecture is

an important aspect for fault emulation. The SEU_Tool uses analytical reasoning and circuit simulation tools to analyze the contribution of combination logic in the path to a sequential or memory element [20]. This tool uses circuit models two levels of simulation at the hardware description language level, and probabilistic models. The SEU_Tool also has algorithms to identify the worst-case contributors to soft errors to reduce computation time. Given the detail of modeling capability in this method, its accuracy is largely a function of the quality and completeness of the parameters used for input [20], [21]. Most tools do not have access to as many inputs as the SEU_Tool, and in particular many micro-architectural fault modeling tools cannot integrate experimental data into the model [22], [23].

In recent years, using architectural simulators have become popular for fault simulation within the computer architecture community. The MARSS and Gem5 [22], [23] simulators have been used to inject faults into the x86 and ARM architectures. These works can inject faults in any part of the simulator, during any cycle and for any duration. Then the output of the simulator can be compared with a golden model so failures can be detected. However, while simulators accurately simulate the instructions, they are generalized and may not accurately represent the a specific processor implementation a user is interested in (e.g. the x86 implementation varies between different families and generations, although each processor can execute x86 code). Therefore, simulation tools can be useful, as long as the modeled architecture is the system being used and the model can be extended to include radiation sensitivities.

Several fault simulation and emulation tools for graphics processing units (GPUs) have been released, including an extension of LLFI for GPUs [24], GPU-Qin [25], SASSIFI [26], GPGPU-sim Fault Injector (GUFI) [27]. GPUs are particularly difficult to model, due to the massively parallel structure of the architecture, so GPU-specific fault injection tools are often necessary for these components. SASSFI has the advantage of being created by the largest manufacturer of GPUs, NVIDIA, and has been adopted by many researchers since its release in 2017.

The tool described in this paper has similarities to both fault simulation and fault emulation tools. The tool provides the observability layer of the software from a software-based fault emulation tool, but has the transistor-level model of fault simulation tools. In that sense, the tool makes it possible to link the faults in the transistor layer to the software layer, so the transition from fault to error can be observed through out the hardware and software architecture. Unlike other fault simulation tools, it is possible to input a number of characteristics, including the radiation test data for the transistors, the software, and the full hardware architecture. The ability to input the full hardware architecture is important, as most simulation tools simulate the instruction set architecture (ISA) only, and do not necessarily accurately depict all the registers that would be implemented in a hardware implementation of the processor. Therefore in our tool, the architectural model is accurate to the hardware architecture, the software model is accurate to the system design, and the radiation sensitivities are accurate to the transistor design and fabrication. This tool has a unique advantage over many other types of fault injection systems, due to level of information that is input into the system.

III. Fault Injection and Tracking

We have a developed a multi-scale simulation framework to understand how transistor level radiation properties can affect a large-scale system. As illustrated in Fig 1, we present a probabilistic framework that goes through four levels of abstraction to model radiation impacts on a running system:

1) The probabilities and physics of an SET on a transistor are modeled to create a probability distribution of charge collected from the radiation event (Section III-A);
2) The charge collected is used to fit a dual, double exponential current source and circuit simulations are used to abstract radiation events as binary SETs on a gate with varying pulse widths [28] (Section III-B);
3) The probabilities and duration of SETs on each gate are abstracted to model the probability of a SEU being latched at the register at the end of the pipeline stage (Section III-A);
4) A discrete event simulation (DES) of software running on a faulty processor is used to simulate and track the faults and the impact of faults on benchmark algorithms are probabilistically characterized. The DES is based on modifying the Structural Simulation Toolkit (SST) [29], a discrete event simulator for high performance computing (HPC) systems (Section III-D).

By using 4 levels of abstraction, we enable a much more comprehensive coverage of possible faults than if Monte Carlo injection were directly used on a full processor model. Fault simulation is focused on the entire system and not just the sensitivity of the microprocessor. Using SST also allows us to model large scale heterogeneous systems with multiple discrete components and to track fault propagation through the system. This modeling framework can be used to both characterize how an existing system will behave in a radiation environment and enable the design of both hardware- and software-level fault mitigation strategies. In the remainder of this section, details about these different layers of abstraction are presented.

A. Radiation Events to Collected Charge

The total charge collected by individual charge generation due to particle strikes in a transistor is determined. We follow the methodology described in [3] consisting of two steps:

- TCAD simulations are used to determine the charge collection efficiencies in each sensitive volume of a transistor;
- Monte Carlo Radiative Energy Deposition (MRED) is used to determine the probability distribution and events with maximum collected charge.

Early soft-error, static-upset rate predictions were calculated with a single rectangular parallelepiped (RPP) sensitive volume using tools that assume the same charge collection rate in the entire volume, vastly over-estimating the soft error

rate (SER) [20]. To overcome this, we use multiple sensitive volumes each with their own charge collection efficiency.

Next, we combine multiple sensitive volumes with Monte Carlo radiation transport techniques to estimate the total charge collected [30], [31]. This tool is known as Monte Carlo Radiative Energy Deposition (MRED). The power of this approach is that it remains tractable in the absence of simplifying assumptions, and therefore in principle, it is more precise and accurate to predict errors. Moreover, it has been experimentally validated [31]–[33].

B. Collected Charge to Single Event Transients

Once the collected charge from a radiation event is determined, we need to convert it to a current source model. Following [28], the collected charge is converted into dual double-exponential current sources. A fast current source provides the charge needed to flip the state and a slow current source models the slow draining of excess charge once the state has been flipped. TCAD models of individual transistors are used to determine the time constants of the current sources. The peak currents of each current source are determined from circuit simulations based on the particular gate type and load and are chosen to flip and hold the output voltage at the opposite rail voltage. Next, the width of the slow current source is determined by setting the integrated charge equal to the collected charge. Finally, the length of rail-to-rail SET is found using a simulation program with integrated circuit emphasis (SPICE) simulation of chained logic gates and only transients that are long enough to propagate and be latched are kept. This process allows us to convert the probability distribution of collected charge to a probability distribution of pulse widths or SEUs for Flip Flops.

C. Logic Gate to Pipeline Stage Fault Simulation

The next level of abstraction is to create a model of SEU probabilities on the output registers of each pipeline stage in a microprocessor. As the effective cross-section is dominated by the registers themselves and not the combinational logic, the cross-section can be approximated as the register cross-section. Nevertheless, we describe how the cross-section can be optionally refined using the probability and pulse width of SETs at individual gates in this section.

It is well known that there are logical masking effects that occur within the circuit which naturally quash some SETs before they are latched [34]. This masking effect is usually calculated for the entire circuit as the circuit reliability. We are interested in the probability of a single SET for any given gate within the circuit becoming logically masked before reaching a register.

The probability of error at register r, for instruction i, P_{ri}, is given by:

$$P_{ri} = P_{R-SEU} + \sum_{\substack{t=test\\vectors}} \sum_{g=gates} P_{TV,ti} \times LM_{tigr} \times P_{G-SEU,tigr} \tag{1}$$

$$P_{G-SEU,tigr} = \sum_{\substack{p=pulse\\widths}} P_{SET_{tigp}} \times \frac{T_{SET_{gpr}}}{T_{clock}} \tag{2}$$

- P_{R-SEU} is the probability of an SEU occurring on the register itself in one clock cycle.
- $P_{TV,ti}$ is the probability of each test vector, t, used for instruction i and (for now) is assumed to be 1/(# test vectors). The possible test vectors for each instruction are randomly sampled.
- LM_{tigr} is 1 or 0 and represents whether an SET on gate g is logically masked at register r based on test vector t and instruction i. It is computed using a static fault simulation with the commercial fault simulator ZO1X.
- $P_{G-SEU,tigr}$ is the calculated probability that an SET at gate g that is not logically masked is latched as an SEU at register r, given test vector t and instruction i.
- $P_{SET_{tigp}}$ is the probability of an SET on gate g with pulse-width p, as computed in Section III. As the SET probability depends on the exact data on the gate (rising vs falling edge), it also depends on the test vector t and instruction i.
- $T_{SET_{gpr}}$ is the modified width of an SET as seen at register r. The radiation induced pulse-width r on gate g needs to be modified by the propagation delay to the register if the rising and falling edges have asymmetric delays and by the average latching window of the register. An average value of the propagation delay, independent of the particular test vector, is used to avoid the need for dynamic fault simulations for every test vector. A reasonable approximation is to directly use the radiation induced pulse width, r, and assume symmetric propagation delays and latching exactly at the clock edge.

To demonstrate the creation of an abstracted model, we consider a simple 32 bit RISC ALU. To highlight the impact of logical masking we assume that an SET on each gate has the same probability, P_{G-SEU}, of being latched as an upset and that a single upset occurs somewhere in the ALU: $P_{G-SEU,tigr} = 1/\#\ of\ gates$. The probability of an error due to the digital logic at each output register of the ALU for an add and or operations is shown in Fig. 3.

D. Pipeline Stage to System Software Fault Simulation

Once the probability of error on each register is computed, faults are injected into a discrete event simulation (DES) of software running on a faulty processor. Software for simulated hardware is written in C or C++, and then compiled into individual assembly instructions that can be run on a DES model of the faulty microprocessor. As shown in Fig. 1, faults are injected into key parts of a processor pipeline. Faults can be injected either as temporary SEUs or as permanent stuck at faults.

A physical processor will have many more registers than fault injection locations shown in Fig. 1. To account for this, the cross-section of each register will be added to one of the modeled fault injection locations. This abstraction allows an architectural simulator to run with a reasonable computational

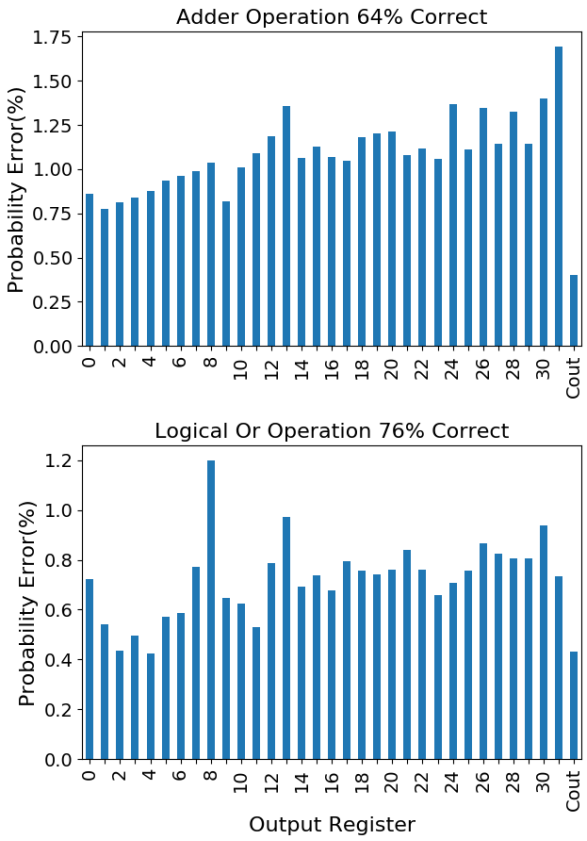


Fig. 3: The probability of an error during ADD and OR operations at each output register due to the digital logic is shown, averaged over one million faults.

efficiency. Accurately accounting for the register level cross-sections from the previous section allows the fault sensitivity of different pipeline stages to be assessed. It also allows the effectiveness of fault mitigation to be determined.

As the DES is based on modifying SST, a discrete event simulator for large scale HPC systems, the fault modeling can encompass the full system, including external memories and other microprocessors. The flexibility of the tool allows each component to be defined individually, such that each component can have its own design, technology node, and radiation sensitivities to a given environment. In this manner, faults can be tracked from component to component, if the fault propagates outside of the component's boundaries. This type of modeling is important with external memory, buses, and multiple processing elements. In cases where the fault causes an SDC within a microprocessor, the microprocessor writes the faulty value into external memory, such as dynamic random access memory (DRAM), or transmits it to another processing element for further calculation. In that sense, faults cascade throughout the system, and the SST tool can track the fault as it moves through a component internally and then transfers to an external component for further propagation.

Faults are inserted into the model by changing the simulator's internal register (reg_word) data structures. This structure is used to represent architectural registers and internal state and is normally a simple 32-bit number. For our simulator, we replace it with a custom data structure that records when the fault occurred, which bit it flipped, and the original fault-free value. Whenever an operation is performed (e.g. adding two registers) we update the faulted and fault-free value and propagate faults to the results register. In this way, we can determine when, where and how a fault is quashed or how it spreads throughout the system to eventually cause a failure. For example, if a register with the value "1" is upset by an SEU in its least significant bit (making it a "0") and is later OR'd with the value "1" it will end up with the correct value as if the SEU never occurred. This sort of "correction by math" can be tracked and counted in the simulator. This capability is useful when designing fault-resistant data encodings or algorithms and is a feature we have not found in other fault simulators.

IV. Case Studies

To show the flexibility and scalability of the tool, we present two case studies. The first one shows how to determine the sensitivity of a D Flip Flop and validates the methodology at the transistor level. The second one uses the D Flip-Flop cross-section to study the effect of SEUs in a pipelined microprocessor architecture while executing different variations of a matrix multiply code.

A. D Flip-Flops

First, we use the tools described above to compute the SEU cross-section of a 14nm D Flip-Flop vs LET and compare the modeling results to experimentally measured cross-sections by Vanderbilt University in [35]. Next, we compute the upset rate of a 14nm D Flip-Flop when exposed to the Adams 90% worst-case environment particles in an isotropic space environment. The upset rate is used as an input in the next processor level case study.

To compute sensitive volumes and charge collection efficiencies, we perform Technology Computer Aided Design (TCAD) SET simulations of a single fin NFET a single fin PFET connected in an inverter configuration as illustrated in Fig. 5. This was done to determine how much of the collected charge would contribute to the SET generation since charge continued to collect over 1 ns. Multiple TCAD simulations are run to simulate particles over a range of linear energy transfers (LETs), and to insert particles at different spots around the fins. The results are mapped into sensitive charge collection volumes that can re-create the collected charge versus deposited charge around the fins. For instance a 3.125 LET particle at normal incidence with a center strike location deposited 0.13 pC but only 0.22 fC were collected. This resulted in a collection efficiency of 0.1%. The charge is injected into a circuit simulation of D Flip-Flop and critical charge for an SEU is determined.

It is also necessary to develop a D Flip-Flop model in the MRED tool. For this model all the fins in the design need sensitive charge collection volumes. Each simulation of an ion/energy results in collected charge at various circuit node(s), which are then compared against critical charges Given a radiation environment, MRED can convert the simulation into

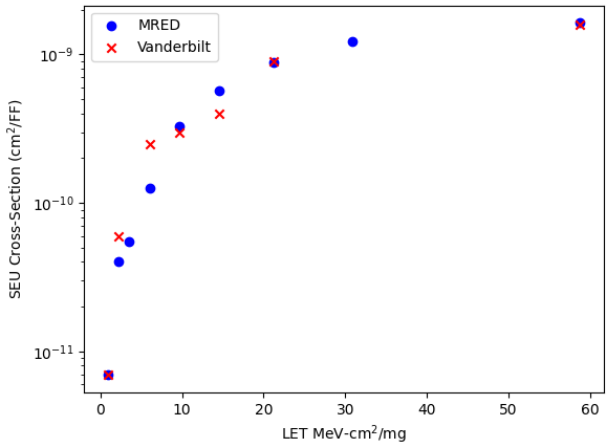
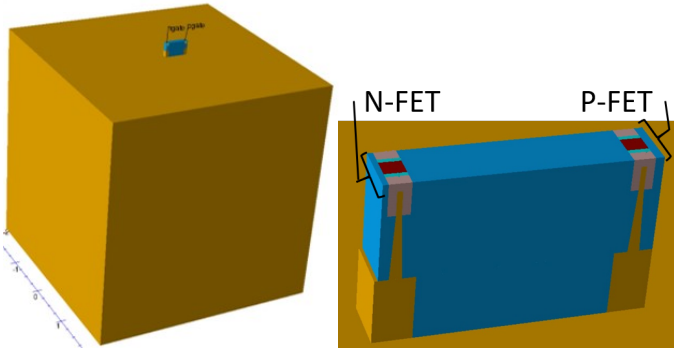


Fig. 4: The experimental and simulated SEU cross-section for a 14nm D Flip-Flop is shown.



(a) View of large substrate (b) Zoomed in view

Fig. 5: TCAD Simulation with (a) showing how large a substrate is needed to allow the charge to recombine and (b) showing a zoomed in view of the fins.

a predicted error rate or cross-section for that environment. To validate our MRED model with test data, we simulated monoenergetic particles at normal incidence matching the test conditions. The resulting cross-section vs LET is plotted in Fig. 4, showing extremely good fit to the experimental data.

Next, the upset rate of a 14nm D Flip-Flop when exposed to the Adams 90% worst-case environment particles in an isotropic space environment is computed to be 4.71×10^{-14} errors/bit/s or 4.07×10^{-9} errors/bit/day. This was estimated by simulating 1×10^{8} particles from the environment and comparing the collected charge resulting from each particle against the critical charges of the transistors as described in [31].

B. MIPS Processor Fault Injection

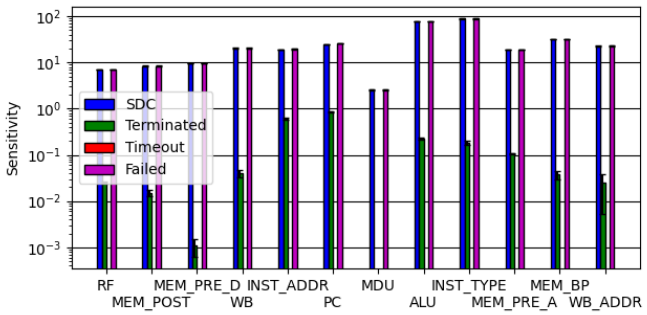
As a case study, we developed a fault simulation model of the clspim MIPS model, based on the R2000 microprocessor, in SST [36].

The simulator models the processor pipeline and so bit-flips that affect the data and memory addresses can be directly modeled. Details of the control flow and exception handling are not modeled and so bit flips that affect those are recorded and the simulation is stopped. These errors would immediately change the operation of the processor and are not logically masked. Faults are injected into the following locations.

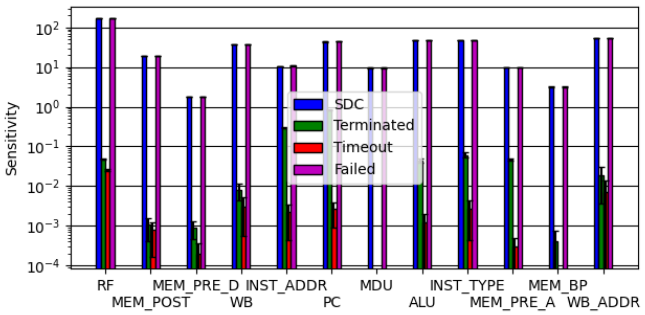
- RF (Register File): Random register in the RF;
- MDU (Multiply-divide Unit): Randomly select a high or low output register of the MDU;
- MEM_PRE_DATA (Memory Data, Pre): Data input to the memory stage (only used if doing a store);
- MEM_PRE_ADDRESS (Memory Address, Pre): Address input to the memory stage;
- MEM_POST (Memory, Post): Output of the memory stage;
- WB (Write Back): Value written back to the register file;
- WB (Write Back "Address"): The location in the register file that data should be written to is changed (i.e. the correct data is written to the wrong place);
- ALU (Arithmetic Logic Unit): Output of the ALU. (This does not impact the output of the MDU but rather it impacts the value sent to the MEM stage and operand forwarding);
- CONTROL_FLOW (Control flow registers): The simulation is halted and a control flow error is recorded;
- EXCEPTION (Exception Handling): If an error occurs in the exception handling hardware, the simulation is halted and an exception error is recorded;
- INSTRUCTION_READ_ADDRESS: The address of the instruction which is fetched from memory;
- INSTRUCTION_TYPE: The instruction itself is modified, potentially causing either the instruction opcode, registers, or immediate values to be incorrectly decoded;
- MEM_BP_VAL (Memory Bypass Value): The value from the memory stage which is forwarded to the ALU stage;
- PC (Program counter): The program counter, indicating the instruction which is next in the program flow;

These locations are shown in the pipeline diagram in Fig. 1. Faults are injected before each cycle of execution either randomly from a probability table or at specified locations, depending on the fault injection mode. For each fault, we can track when it is injected, how it spreads throughout the system, and if/when it gets masked or quashed (corrected).

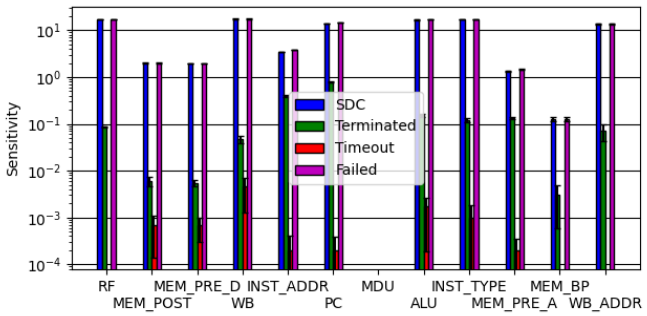
In order to accurately compute the cross-section we need to compute how many registers contribute to each fault injection location. To do this, and to eventually allow for comparison with a beam test, we analyzed the HERMES microprocessor [37], which is a MIPS 4KC based processor that has dual redundant processor pipelines and triple redundant configuration. This allows for an SEU to be localized to a particular part of the processor. First, all the registers in the dual redundant pipeline were listed and attributed to one of the fault injection locations above. This allows us to capture the impact registers that are not directly modeled in an architectural simulator. We choose to disable the caches to simplify the modeling. This gives a model that can be correlated to future beam tests of HERMES. The HERMES processor does contain buffers for the caches and bus I/O queues, which use custom self-



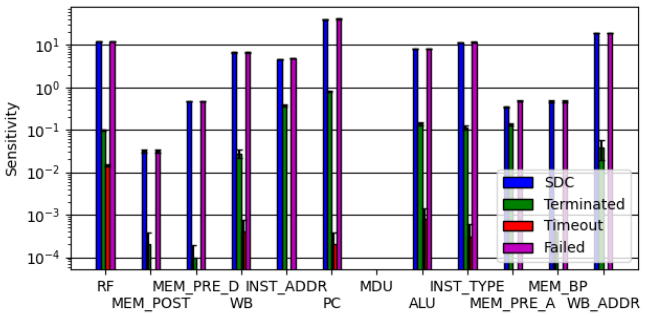
(a) Matrix Multiply (Simplex)



(b) Matrix Multiply O3 Optimizations



(c) With DMR O3



(d) With TMR O3

Fig. 6: Failure sensitivity of different microprocessor registers to faults with correct execution percentages shown beneath. CONTROL and EXCEPTION (not shown) = 100%

correcting TMR flip-flops so they are correct in the event of a DMR recovery. We modeled these as latches whose cross-sections would be almost equivalent to a FF. For the purposes of this paper, we added parts of the processor that are protected in TMR so that the results will be representative of a typical processor. However, details of the bus protocol, the co-processor, and the division operation were not modeled. Additionally, caches are disabled and the simulation is stopped after exception handling and control flow errors. In Table 1, we give the number of register bits that contribute to the cross-section of each fault injection location, representing the number of physical bits in our processor (as opposed to number of bits in our simulation registers). This information, not usually provided by other fault simulators, is needed to eventually calculate the cross-section. For the MDU, the register count is multiplied by the number of cycles the MDU is active for as a detailed MDU is not modeled.

To summarize the main differences between the MIPS SST model and HERMES:

- SST lacks a TLB, but the TLB can be turned off in HERMES, removing its cross-section for correlation purposes;
- The SST model does not perform cache request coalescing;
- The SST model does not have the recovery pipeline stage which is only used in HERMES during a soft-error exception.

This case study specifically looks at an integer matrix-matrix multiply benchmark and a quicksort benchmark [38].

Location	Bit	Location	Bit
ALU	64	CONTROL	52
INST_ADDR DATA	40	INST_ADDR CTRL	16
INST_TYPE DATA	32	INST_TYPE CTRL	5
MDU DATA	233	MDU CTRL	122
MEM_POST DATA	160	MEM_POST CTRL	8
RF DATA	992	RF CTRL	5
MEM_PRE_ADDR DATA	203	MEM_PRE_ADDR CTRL	48
MEM_PRE_DATA DATA	256	MEM_PRE_DATA CTRL	8
MEM_PRE_ADDR BYTE	16	MEM_PRE_DATA BYTE	23
MEM_BP_VAL DATA	32	WB	32
PC	64		

TABLE I: The number of register bits contributing to each fault injection location is given. DATA errors result in a single bit being flipped and CONTROL errors result in all bits being randomized. Byte errors result in eight bits being randomized.

To demonstrate the versatility of the simulation infrastructure, the same benchmarks are implemented [39] in different ways:

- Matrix-Matrix Multiply
  - Simplex: A standard matrix multiplication of two 12x12 32-bit unsigned integer matrices using a triple-nested loop.
  - Dual Modular Redundancy (DMR): The operations in the innermost loop are duplicated and results are compared. If the outputs do not match, the computation is performed again, so that results can be voted.
  - Triple Modular Redundancy (TMR): Each multiplication operation is completed three times and a bitwise

majority vote determines the output. The load operation for the multiply is also triplicated to ensure that the registers are not shared.
  - RD: The multiplication uses reduced-precision multiplication using 16-bit integers.
- Quicksort (qsort)
  - Simplex: A standard integer recursive quicksort sorting algorithm which sorts a list of 100 integers four times (twice forwards, twice backwards)
  - RD: The comparison uses reduced-precision 16-bit integers

To explore the effect of compiler optimizations, each of these variants is compiled with and without compiler optimizations for a total of eight variations of the code. Code optimization often focuses on three different types of optimizations: 1) preferentially using local registers instead of cache memory and 2) unrolling loops to reduce branching that could lose performance on an improperly predicted branch, and 3) reusing intermediate results, such as address offsets, to reduce recomputation.

These optimizations can affect the radiation sensitivity of the software in unexpected manners [40], [41]. If the registers are more SEU sensitive than the cache, then it is possible to increase the radiation sensitivity of the software by using more registers. Likewise, avoiding branching hardware and reusing computation, reduces the affect of SEUs and SETs in the control and mathematical units. The performance of the software can also affect the overall radiation sensitivity [41]. In many cases, faster software is more resilient software, because the radiation exposure is minimized. In this case, the optimized code runs in about one quarter of the time of the unoptimized code, so the radiation exposure is reduced by one quarter. Because these results can increase and decrease the radiation sensitivity of the software, the ability to quickly test these different variations is helpful in determining which of these issues will dominate the radiation sensitivity [41].

Each executable was run 384,000 times in SST on the MIPS architecture model. Each simulation took between .4 and 1.6 seconds dependent on the compilation options and the fault injection (e.g. an early fault injection could cause a crash and cause the simulation to end very quickly). During each test one SEU-type fault is injected into a fault location during the execution. The timing of the injection is chosen randomly. The fault location is weighted by the computed cross-section. The execution was categorized into one of five states:

1) SDC (Silent Data Corruption): The program completes, but the results are incorrect.
2) Terminated: The program failed to complete, which is often from executing an illegal memory operation.
3) Timeout: The test's execution time is more than four times the expected execution time, which is often an indication that the test is stuck in an infinite loop;
4) Stopped: Control Flow and Exception errors result in the simulation being stopped. These errors are added to the total failed error rate.
5) Correct: The program completes in the expected amount of time without an error signal, and the results are correct. Inverse of plotted failed bar.

The results of our fault injection study are shown in Fig. 6.

The different algorithmic variants of the basic benchmark have a substantial effect on failure sensitivity. As expected, the TMR and DMR versions eliminate any risk of MDU faults and the RD version reduces MDU sensitivity by about half. TMR/DMR also substantially reduces the risk of SDC errors from other faults, while having less of an impact on Terminated or Timeout errors.

Compiler optimization also impacts the failure sensitivity. The greater use of registers makes the optimized version 3-11$\times$ more susceptible (on a per-fault basis) to Register File (RF) faults and up to 2$\times$ more susceptible to MEM_POST faults. Overall, an individual fault is more likely to cause a failure or SDC in the optimized versions. However, because the optimized code runs so much faster, their susceptibility over time is much less. These results show the versatility of our tool and the type of data we can collect.

In addition to measuring the sensitivities of different failure modes in registers, we can collect many other statistics, presented in Figs. 7-10 for only the matmat simplex program with O3 optimizations. Fig. 7 shows how many instructions were executed after a fault injection until the processor had some sort of failure. Intuitively, injecting an SEU in the address of an instruction or data load has an almost immediate impact. With the exception of MEM_PRE_DATA, most injections cause failure within about 100 instructions. This suggests that for this benchmark, if an SEU is going to cause a failure, it will do so quickly.

Fig. 8 shows how often we injected into a register that was not actively being used. Since the register is not in active use the fault had no affect on the program execution or processor state. The PC and RF are always in use and are thus never architecturally masked. With the exception of those registers, every other register has a masking factor of .5 or more, suggesting that SEUs in these registers can have no effect over 50% of the time. In future studies we can look at these masking factors over numerous benchmark programs to observe if this holds true to for other common algorithms.

Fig. 9 shows how often a fault gets masked through math operation occurring in the natural program execution. This can happen when only the most significant bits are used or when something is multiplied by zero, etc. We note that for a fault to be corrected, all the bits must be correct (even if the least significant bits are in the noise and are thus unimportant for "correct" program execution). Intuitively, math corrections should rarely occur since the math operation needs to be masking bits and the fault must be in the bits being masked. RF fault are the most likely to be corrected since they are the most likely to be immediately used in a math operation by the ALU or MDU.

Fig. 10 shows how faults spread throughout the system. Injections in some registers such as the instruction address have an almost immediate impact on the program control, causing a crash/exception almost immediately, so there are not many additional faulted registers. On the flip-side, faults in the register file usually affects data and not the control flow. If the register is used in many other calculations the fault can

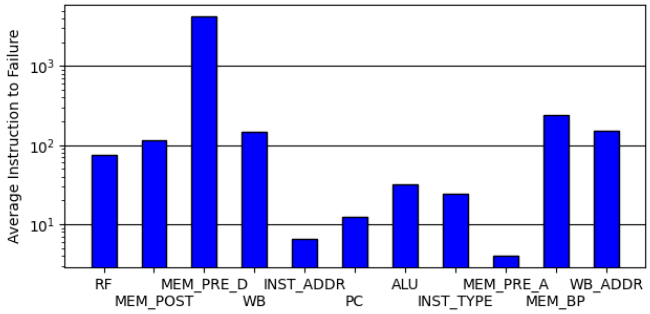


Fig. 7: The average number of instructions after a fault injection until the processor failed.

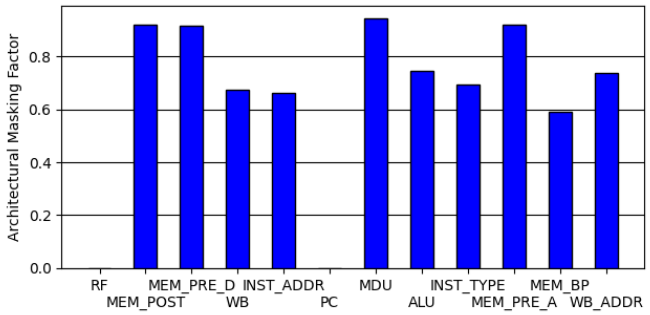


Fig. 8: Fraction of injections into components which were not in use during that injected clock cycle for matmatO3.

spread quickly, which then affects other registers used in other calculations causing an exponential growth in the faults spread.

Using the data collected in fault injection for all the registers combined with the register sizes in Table I, we can determine the composite failure rate of the processor, shown in Fig. 11. The composite failure rate is computed by taking a weighted average of each register's failure rate (weight is determined by percentage of total bits represented by that register). This gives us the probability of a failure from a random SEU within any register in the processor.

With this probability we can calculate the cross-section in a given environment using the per-bit cross-section and the number of bits in the processor, presented in Fig. 11. This is computed using a simple multiplication of the probability

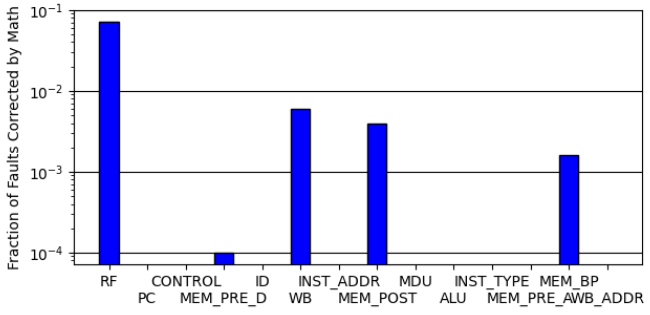


Fig. 9: Fraction of faults that were corrected through math operations during normal program execution for matmatO3.

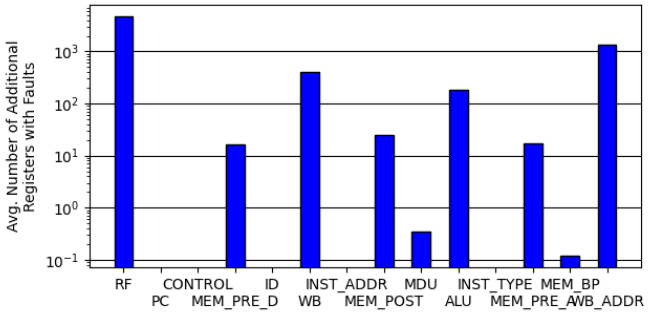


Fig. 10: Average number of registers corrupted during program execution after the fault injection for matmatO3.

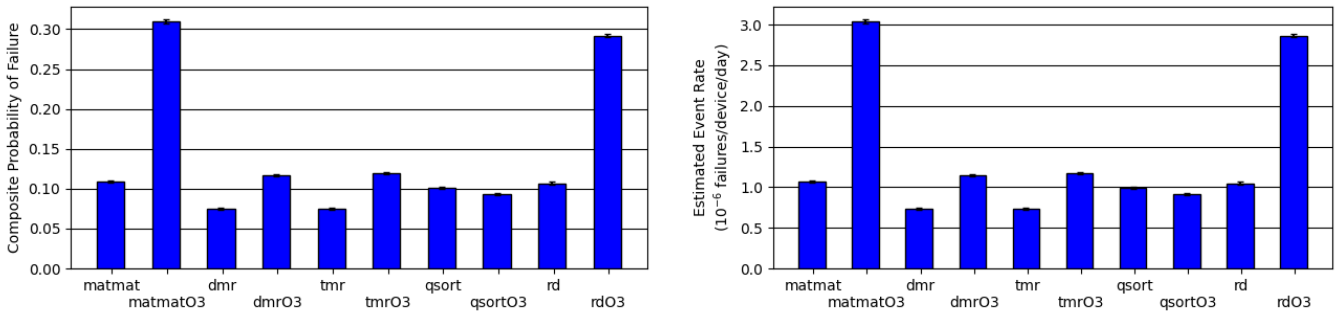
of failure, per-bit cross-section and total number of bits in the processor. We did this calculation using the Adams 90% worst-case environment in an isotropic space environment. Recall, the SEU rate of this environment was calculated in section IV-A.

These calculations assume that the algorithm runs on an infinite loop. We then scale these calculated cross-sections based on program execution time (in number of clock cycles), relative to the matmat simplex program and present those results in Fig. 12. DMR and TMR versions of the matmat perform worse in this scaled version because they they increase execution time more than reducing the probability of failure. On the flip-side, the O3 versions perform better since they reduce execution time more than increasing the failure probability. Interestingly, the DMR O3 gives the lowest failure probability. This is likely because the DMR provides some mitigation without increasing the number of execution clock cycles drastically. Then, the O3 optimization flag is able to make the most efficient use of these mitigations.

V. Conclusion

As computing systems become larger and more complex, it becomes increasingly difficult, but important to model and understand how faults induced by SEEs propagate. Faults may get masked through natural flow of the program, or they could sit idle and then cause havoc long after they first appeared in the system. Understanding their behaviors will be essential for developing future state-of-the-art mitigation strategies and will help us better understand data collected through more typical irradiation experiments.

We have extended upon fault simulation work by directly correlating simulation registers with physical registers of a actual hardware netlist and by allowing the injection rate of the simulator to be dependent on radiation environment data. This allows us to estimate cross-section rates based on the fault injection results and environment. We have demonstrated a multi-scaled framework for injecting, tracking and analyzing faults in a micro-processor based system. Most of this framework will eventually be made open-source and available to the general community. We have incorporated physics based calculations to estimate SEU and SET cross-sections by using TCAD simulations using a chosen process technology and logical/architectural masking effects can be calculated using the circuit's netlist. We have shown these results using the



(a) Probability of failure

(b) Estimated cross-section

Fig. 11: (a) Probability of error given a bit error. Each registers probability of being fault injected was scaled according to it's size and logical masking. (b) Estimated cross-section using Adams 90% worst-case environment in an isotropic space environment.

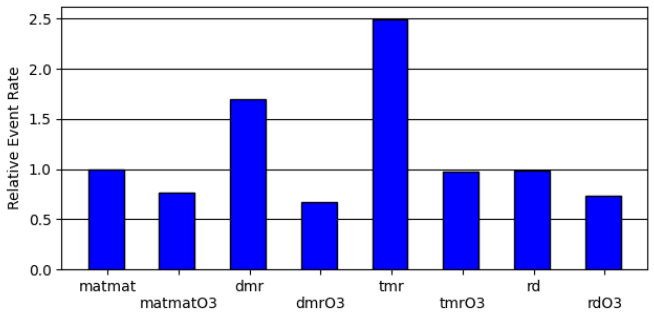


Fig. 12: Estimated cross-section scaled to program run-time (execution cycles/matmat execution cycles)

Adams 90% worse-case environment in an isotropic space environment, but with the framework we have built, we can estimate cross-sections in many other environments. The collected SEU/SET and masking data can be used to drive fault injection on high performance simulators, simulating the actual hardware, where the fault can be monitored for disruptive behavior.

REFERENCES

[1] P. E. Dodd, M. R. Shaneyfelt, J. R. Schwank, and J. A. Felix, "Current and future challenges in radiation effects on CMOS electronics," *IEEE Transactions on Nuclear Science*, vol. 57, no. 4, pp. 1747–1763, Aug 2010.

[2] R. Velazco, Dale McMorrow, and J. Estela, Eds., *Radiation Effects on Integrated Circuits and Systems for Space Applications*. Springer, 2019.

[3] H. M. Quinn, D. A. Black, W. H. Robinson, and S. P. 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.

[4] J. Carreira, H. Madeira, and J. G. 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.

[5] 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.

[6] B. Nicolescu, P. Peronnard, R. Velazco, and Y. Savaria, "Efficiency of transient bit-flips detection by software means: a complete study," in *Proceedings 18th IEEE Symposium on Defect and Fault Tolerance in VLSI Systems*, 2003, pp. 377–384.

[7] P. A. Ferreyra, C. A. Marques, R. Velazco, and O. Calvo, "Injecting single event upsets in a digital signal processor by means of direct memory access requests: a new method for generating bit flips," in *RADECS 2001. 2001 6th European Conference on Radiation and Its Effects on Components and Systems (Cat. No.01TH8605)*, 2001, pp. 248–252.

[8] J. Wei, A. Thomas, G. Li, and K. Pattabiraman, "Quantifying the accuracy of high-level fault injection techniques for hardware faults," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014, pp. 375–382.

[9] J. Cong and K. Gururaj, "Assuring application-level correctness against soft errors," in *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2011, pp. 150–157.

[10] V. C. Sharma, A. Haran, Z. Rakamaric, and G. Gopalakrishnan, "Towards formal approaches to system resilience," in *2013 IEEE 19th Pacific Rim International Symposium on Dependable Computing*, 2013, pp. 41–50.

[11] M. de Kruijf, S. Nomura, and K. Sankaralingam, "Relax: An architectural framework for software recovery of hardware faults," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 497–508. [Online]. Available: https://doi.org/10.1145/1815961.1816026

[12] K. M. Zick, C. Yu, J. P. Walters, and M. French, "Silent data corruption and embedded processing with NASA's SpaceCube," *IEEE Embedded Systems Letters*, vol. 4, no. 2, pp. 33–36, 2012.

[13] M. Bucciero, J. P. Walters, R. Moussalli, S. Gao, and M. French, "The PowerPC 405 memory sentinel and injection system," in *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*, 2011, pp. 154–161.

[14] Junjie Peng, Jun Ma, Bingrong Hong, and Chengjun Yuan, "Validation of fault tolerance mechanisms of an onboard system," in *2006 1st International Symposium on Systems and Control in Aerospace and Astronautics*, 2006, pp. 1230–1234.

[15] M. Rebaudengo and M. Sonza Reorda, "Evaluating the fault tolerance capabilities of embedded systems via BDM," in *Proceedings 17th IEEE VLSI Test Symposium (Cat. No.PR00146)*, 1999, pp. 452–457.

[16] A. V. Fidalgo, G. R. Alves, and J. M. Ferreira, "A modified debugging infrastructure to assist real time fault injection campaigns," in *2006 IEEE Design and Diagnostics of Electronic Circuits and systems*, 2006, pp. 172–177.

[17] J. Isaza-González, A. Serrano-Cases, F. Restrepo-Calle, S. Cuenca-Asensi, and A. Martínez-Álvarezs, "Dependability evaluation of COTS microprocessors via on-chip debugging facilities," in *2016 17th Latin-American Test Symposium (LATS)*, 2016, pp. 27–32.

[18] S. Mirkhani, H. Cho, S. Mitra, and J. A. Abraham, "Rethinking error injection for effective resilience," in *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2014, pp. 390–393.

[19] K. S. Yim, Z. Kalbarczyk, and R. K. Iyer, "Measurement-based analysis of fault and error sensitivities of dynamic memory," in *2010 IEEE/IFIP International Conference on Dependable Systems Networks (DSN)*, 2010, pp. 431–436.

[20] L. W. Massengill, A. E. Baranski, D. O. Van Nort, J. Meng, and B. L. Bhuva, "Analysis of single-event effects in combinational logic-simulation of the AM2901 bitslice processor," *IEEE Transactions on Nuclear Science*, vol. 47, no. 6, pp. 2609–2615, 2000.

[21] V. Srinivasan, A. L. Sternberg, A. R. Duncan, W. H. Robinson, B. L. Bhuva, and L. W. Massengill, "Single-event mitigation in combinational logic using targeted data path hardening," *IEEE Transactions on Nuclear Science*, vol. 52, no. 6, pp. 2516–2523, 2005.

[22] K. Parasyris, G. Tziantzoulis, C. D. Antonopoulos, and N. Bellas, "GemFI: A fault injection tool for studying the behavior of applications on unreliable substrates," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2014, pp. 622–629.

[23] M. Kaliorakis, S. Tselonis, A. Chatzidimitriou, N. Foutris, and D. Gizopoulos, "Differential fault injection on microarchitectural simulators," in *2015 IEEE International Symposium on Workload Characterization*, Oct 2015, pp. 172–182.

[24] G. Li, K. Pattabiraman, C. Cher, and P. Bose, "Understanding error propagation in GPGPU applications," in *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016, pp. 240–251.

[25] B. Fang, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi, "GPU-Qin: A methodology for evaluating the error resilience of GPGPU applications," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014, pp. 221–230.

[26] S. K. S. Hari, T. Tsai, M. Stephenson, S. W. Keckler, and J. Emer, "SASSIFI: An architecture-level fault injection tool for GPU application resilience evaluation," in *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2017, pp. 249–258.

[27] S. Tselonis and D. Gizopoulos, "GUFI: A framework for GPUs reliability assessment," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2016, pp. 90–100.

[28] D. A. Black, W. H. Robinson, I. Z. Wilcox, D. B. Limbrick, and J. D. Black, "Modeling of single event transients with dual double-exponential current sources: Implications for logic cell characterization," *IEEE Transactions on Nuclear Science*, vol. 62, no. 4, pp. 1540–1549, Aug 2015.

[29] A. F. Rodrigues *et al.*, "The structural simulation toolkit," *SIGMETRICS Perform. Eval. Rev.*, vol. 38, no. 4, p. 37–42, Mar. 2011. [Online]. Available: https://doi.org/10.1145/1964218.1964225

[30] A. D. Tipton *et al.*, "Multiple-bit upset in 130 nm CMOS technology," *IEEE Transactions on Nuclear Science*, vol. 53, no. 6, pp. 3259–3264, Dec 2006.

[31] R. A. Weller *et al.*, "Monte carlo simulation of single event effects," *IEEE Transactions on Nuclear Science*, vol. 57, no. 4, pp. 1726–1746, Aug 2010.

[32] K. M. Warren *et al.*, "Heavy ion testing and single event upset rate prediction considerations for a DICE flip-flop," *IEEE Transactions on Nuclear Science*, vol. 56, no. 6, pp. 3130–3137, Dec 2009.

[33] R. A. Weller *et al.*, "General framework for single event effects rate prediction in microelectronics," *IEEE Transactions on Nuclear Science*, vol. 56, no. 6, pp. 3098–3108, Dec 2009.

[34] D. T. Franco, M. C. Vasconcelos, L. Naviner, and J.-F. Naviner, "Signal probability for reliability evaluation of logic circuits," *Microelectronics Reliability*, vol. 48, no. 8, pp. 1586 – 1591, 2008, 19th European Symposium on Reliability of Electron Devices, Failure Physics and Analysis (ESREF 2008).

[35] R. C. Quinn *et al.*, "Heavy ion SEU test data for 32nm SOI flip-flops," in *2015 IEEE Radiation Effects Data Workshop (REDW)*, 2015, pp. 1–5.

[36] A. Rogers and S. Rosenberg, "Cycle level SPIM," *Dept. Comput. Sci., Princeton Univ., Princeton, NJ*, July 1993.

[37] L. T. Clark, D. W. Patterson, C. Ramamurthy, and K. E. Holbert, "An embedded microprocessor radiation hardened by microarchitecture and circuits," *IEEE Transactions on Computers*, vol. 65, no. 2, pp. 382–395, 2016.

[38] H. Quinn *et al.*, "Using benchmarks for radiation testing of microprocessors and FPGAs," *IEEE Transactions on Nuclear Science*, vol. 62, no. 6, pp. 2547–2554, 2015.

[39] A. Rodrigues, "benchmark_code forked github repository," https://github.com/afrodri/benchmark_codes/releases/tag/nesrec2020, October 2020.

[40] A. Lindoso, L. Entrena, M. García-Valderas, and L. Parra, "A hybrid fault-tolerant LEON3 soft core processor implemented in low-end SRAM FPGA," *IEEE Transactions on Nuclear Science*, vol. 64, no. 1, pp. 374–381, 2017.

[41] F. M. Lins, L. A. Tambara, F. L. Kastensmidt, and P. Rech, "Register file criticality and compiler optimization effects on embedded microprocessor reliability," *IEEE Transactions on Nuclear Science*, vol. 64, no. 8, pp. 2179–2187, 2017.