SASSIFI: An Architecture-level Fault Injection Tool for GPU Application Resilience Evaluation

Siva Kumar Sastry Hari, Timothy Tsai, Mark Stephenson, Stephen W. Keckler, Joel Emer
NVIDIA

***Abstract*—As GPUs become more pervasive in both scalable high-performance computing systems and safety-critical embedded systems, evaluating and analyzing their resilience to soft errors caused by high-energy particle strikes will grow increasingly important. GPU designers must develop tools and techniques to understand the effect of these soft errors on applications. This paper presents an error injection-based methodology and tool called SASSIFI to study the soft error resilience of massively parallel applications running on state-of-the-art NVIDIA GPUs. Our approach uses a low-level assembly-language instrumentation tool called SASSI to profile and inject errors. SASSI provides efficiency by allowing instrumentation code to execute entirely on the GPU and provides the ability to inject into different architecture-visible state. For example, SASSIFI can inject errors in general-purpose registers, GPU memory, condition code registers, and predicate registers. SASSIFI can also inject errors into addresses and register indices. In this paper, we describe the SASSIFI tool, its capabilities, and present experiments to illustrate some of the analyses SASSIFI can be used to perform.**

1. Introduction

Transient hardware errors caused by high-energy particle strikes are a rising concern for processors deployed in large-scale systems and safety-critical embedded systems. These transient errors can propagate to the application level and cause execution failures, also known as Detected Unrecoverable Errors (DUEs), or silently corrupt application output producing Silent Data Corruptions (SDCs). Since SDCs are silent, developing insights to what causes them and designing cost-effective mitigation schemes are important and challenging.

SDCs are fundamentally application dependent. Application-level error injection techniques with the ability and speed to run many error injection simulations to application completion are needed to evaluate their effect on the program output. Such techniques will allow us to analyze which type of errors in different application sections produce SDCs. Until recently, the tools that allow us to perturb and monitor GPU state at the architecture level in an automated, efficient, and user-friendly manner have not been available. A recent tool called SASSI overcomes these challenges and provides the ability to instrument instructions in the low-level GPU assembly language (SASS) [1]. The instrumentation allows callbacks to arbitrary user-level functions which can execute before or after the instrumented instructions. This paper demonstrates how SASSI can be employed to evaluate and analyze GPU application resilience to transient errors by injecting errors into the architecture state on a running GPU. We call this tool SASSIFI (SASSI-based Fault Injector).

SASSIFI provides an automated flow to perform error injection campaigns. It operates in three main steps: (1) profiling and identifying the error injection space; (2) statistically selecting error injection sites; and (3) injecting errors into executing applications and monitoring error behavior. Steps 1 and 3 use SASSI instrumented applications.

For step 1, we collect the names of the kernels and the number of times they execute in an application using CUPTI. CUPTI allows host-side code to receive callbacks when certain CUDA events such as kernel launches and exits occur [2]. We instrument before all instructions using SASSI and inspect the register and memory information to count the number of dynamic instructions of different instruction types. Using this information, SASSIFI selects a statistically significant number of error injection sites based on the desired confidence interval. For each injection, we identify a tuple that contains enough information to identify *when* and *what* error to inject. As selecting the injection model depends on the type of studies one wants to perform, SASSIFI provides several options described below.

In the last and most important step, SASSIFI injects one error per application run and monitors for a crash, hang, failure symptom, and output corruption. In each injection run, we check whether the selected injection site (as described by the tuple) has been reached. We do so by monitoring the executing kernels and assembly instructions through CUPTI and SASSI instrumentation handlers, respectively. At the injection site, we inject the error through the instrumentation handler either before or after the instruction, based on the error indicated in the tuple.

SASSIFI is publicly available on GitHub at https://github.com/NVlabs/sassifi. It can be used to perform the following two types of studies. (1) Inject bit-flips into the register file, randomly spread across time and space, to compute the Architectural Vulnerability Factor (AVF) of the register file, which informs us of the importance of using ECC on this structure. (2) Inject errors in the outputs of the instructions to analyze how a low-level soft error that manifests at the architecture level can propagate to the

program output.

For the first type of the study, SASSIFI can inject single and double bit-flips in randomly selected registers at a randomly selected dynamic instruction to capture the impact of direct particle strikes in the register file. For the second type of study, SASSIFI injects errors into the outputs of the dynamic instructions (e.g., destination register value). This approach allows quantification of the probability of application-level SDCs given that a low-level soft error manifests as bit-flips in the outputs of the instructions. Since low-level errors can manifest in different ways at the architecture level, SASSIFI provides the ability to inject errors into the output values and addresses of the instructions. SASSIFI can be used to inject errors into any architecturally visible state including general-purpose registers, condition codes, predicate registers, and store addresses/values. SASSIFI can also be used to select a group of instructions for targeted studies such as analyzing whether errors in floating-point operations cause more SDCs than the integer operations, which can be helpful in designing cost-effective error detection schemes.

SASSIFI can also inject a variety of errors in the output values and addresses. For example, SASSIFI can inject single and double bit-flips, as well as random and zero values in the output of one instruction in one thread. It can also inject these error patterns in the same instruction in all the threads in a warp (group of threads).

Leveraging the capabilities of SASSIFI, we performed several case studies and present a summary of the results using two DOE mini-apps (Lulesh and CoMD) and all workloads from the Rodinia benchmark suite [3], [4], [5]. Our findings are as follows.

- By simulating particle-strikes in the register file, we show that enabling SECDED ECC yields limited resilience benefits for some applications, but is required for others. SASSIFI can be used to identify the workloads that might not need protection and can save energy by disabling ECC.
- Error injections into destination registers of instructions that write to general-purpose registers show that only a small fraction result in SDCs. The SDC probability is application-dependent, and the per-kernel results provide insights for developing cost-effective SDC mitigation schemes. For example, duplicating just one kernel in the backprop workload can eliminate most of the SDCs.
- Error injections into different instruction groups show that errors in some groups are less likely to produce SDCs for a given application. For example, double-precision floating-point operations in CoMD are much less susceptible than integer operations. Targeting error mitigation schemes only for the high SDC susceptible instruction groups may result in a cost-effective solution.
- Injecting different types of bit-flip patterns for the same group of instructions as well as injecting into addresses versus values reveal that the SDC probabilities depend on the chosen bit-flip model and the architecture state for corruption. This observation makes understanding how low-level errors manifest at the architecture level important for accurate SDC rate estimates.

2. SASSIFI Framework

2.1. SASSI Background

SASSI is a compiler-based instrumentation tool that runs as the final pass in NVIDIA's production backend compiler and assembler, `ptxas` [1]. Because SASSI is invoked after the original, un-instrumented SASS has already been finalized, the injected instrumentation does not disrupt the perceived final instruction schedule or register usage. In this paper, we use SASSI to inject instrumentation after all SASS instructions that modify registers or memory.

SASSI must be instructed *where* to insert instrumentation and *what* instrumentation to insert. For each of the instrumentation sites, SASSI will insert a CUDA ABI-compliant function call to a user-defined *instrumentation handler* function, passing site-specific information as arguments to the handler. Therefore, users must instruct SASSI *what* information to pass to the instrumentation handler(s). We can currently extract and pass to an instrumentation handler the following information for each site: memory information (e.g., addresses read and written), register usage information (e.g., registers read and written, including their values), conditional branch information, and register liveness information.

Unlike CPU instrumentation, GPU instrumentation must coordinate with the host CPU to both initialize instrumentation counters and to gather their values. We use the CUPTI library to initialize counters before kernels launch and to copy information off the device after kernels exit.

2.2. SASSIFI Tool Description

SASSIFI is an automated tool that can perform error injection campaigns. It operates in three main steps: (1) profiling and identifying the error injection space; (2) statistically selecting error injection sites; and (3) injecting errors into executing applications and monitoring error behavior. Steps (1) and (3) occur on different executions of the SASSI instrumented application on the GPU; step (2) is performed on the host CPU. We explain these steps below using Figure 1.

For any error injection campaign, one needs to specify an error model that defines *what* errors to inject and *where* (in time and space) to inject them in a program. One example of an error model is injecting a single bit-flip error (*what*) in destination registers of randomly selected floating-point instructions (*where*). The three SASSIFI steps take the error model as input to perform the injection campaign.

Step 1: Profiling and identifying the error injection space. We collect the information needed to identify the error injection space in this step. Specifically, we collect (1) static kernel names in an application, (2) the number of times each kernel executes, and (3) the number of dynamic instructions per instruction group (a set of opcodes), as

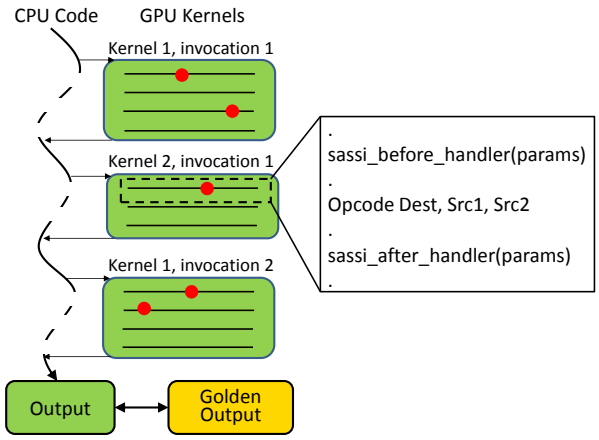

Figure 1: SASSIFI overview.

described by *where* in the error model. We only count the instructions that are executed. We use CUPTI to collect (1) and (2) and instrument the GPU instructions using SASSI to collect (3). We instrument before all instructions. We pass the register and memory information of the instrumented instruction to the handler to determine whether the instruction matches the *where* description from the error model. We count the number of dynamic instructions if the match is successful. For the example error model mentioned above, this step counts the number of floating-point instructions that write to a register. We count the dynamic instructions separately for each of the instruction groups as described by the input error models.

Step 2: Statistically selecting error injection sites. Using the information from step 1, SASSIFI selects a statistically significant number of error injection sites per application for a desired confidence interval. The red dots in Figure 1 show a set of randomly selected error sites in an application that has two static and three dynamic kernels. We define an error site as a tuple that captures enough information needed to select a specific dynamic instruction from the program within the selected instruction group (*where*) and how that selected instructing is to be perturbed (*what*). For the above example, an error site will specify a randomly selected dynamic instruction number among all the floating-point instructions that write to a register in the program. It also specifies that a single-bit flip is to be injected and randomly selects a destination register in that instruction.

Since we know the dynamic instruction count breakdown per kernel invocation from the profiling phase, we combine the instructions from all the kernel executions (for each of the instruction groups) and randomly select dynamic instruction numbers for error injections. We map this dynamic instruction number back to a static kernel name and dynamic kernel invocation index, which become the part of the error site tuple. We also record the dynamic instruction count for the selected instruction group relative to the beginning of the current kernel in the tuple. For *what*, we select an instruction's output (if the instruction has multiple outputs) and the error value to inject in that output. For this purpose, we generate and include two random numbers in the tuple.

Step 3 Error injections runs. In the last and most important step, we inject one error per application run and monitor for crashes, hangs, failure symptoms, and output corruption. In each injection run, we check whether the selected static kernel and dynamic kernel count has been reached using CUPTI on the host CPU. If so, we copy the remaining error site tuple into the device memory. During kernel execution, we maintain a counter for the selected instruction group and check whether the dynamic instruction count that will execute next or just executed (based on whether the check is performed in the SASSI handler before or after the instruction, respectively) is the selected instruction. If so, we inject the error at this instruction by selecting the register and the location of the bit-flip using the two random numbers passed by the tuple. In Figure 1, this process corresponds to injecting each of the red dots one at a time in separate runs and comparing the output to the golden output, in the absence of failures.

Based on the selected error model, SASSIFI instruments either before, after, or both before and after the instructions. For example, to inject errors in a randomly selected register at a randomly selected instruction (used to measure register file AVF), we instrument the instructions before they execute. For injecting errors in a destination register of an executing instruction (used to represent errors in unprotected pipeline bits), we instrument after the instruction. Finally, to inject an address or register index error, we instrument instructions both before and after the instruction.

After error injection, the application is resumed where it then executes to completion, unless a crash or a hang is detected. We categorize the injection outcome based on the exit status of the application, hang detection, error messages thrown during execution, and differences in *stdout/stderr* and program output (typically stored in a file) from that of an error-free execution of the program. We define the outcome where the execution either terminates early with non-zero exit status or hangs as a *DUE* (Detected Uncorrectable Error). Whenever we observe error messages in either the system log (using the Linux utility *dmesg*), *stderr*, or *stdout*, we categorize the outcome as a *potential DUE*.[1] If the application finishes execution and the application output matches the expected output and the run is not categorized as DUE or potential DUE, we categorize it as *masked*. If the output differs from the expected output, we categorize it as *SDC*. We explain these outcome categories in Table 1.

During injection runs, SASSIFI can also track whether the injected error was ever consumed before being overwritten by monitoring the source and destination register numbers of the executing instructions using a SASSI handler. This information provides better understanding into why certain errors are masked and also provides an opportunity to terminate early and optimize the injection runtime. We explore this option in one of the use cases described below.

1. A run that is characterized as *potential DUE* will result in either an SDC or masked outcome if the appropriate symptom monitors are not present in the system. SASSIFI distinguishes such events, but we do not present that breakdown here for brevity.

TABLE 1: Error injection outcomes.

Category	Explanation
Masked	Application output is same as the error free output. No error symptom is observed.
DUE	Application exits with non-zero exit status. Execution does not terminate within an allocated threshold time, which is 3× the fault-free runtime in our study.
Potential DUE	Unsuccessful kernel executions (detected by comparing kernel exit status with *cudaSuccess*) or explicit error messages in *stdout/stderr* (e.g., Error: misaligned address). These errors can be categorized as detected if the system has appropriate application or system monitors.
SDC	Application finishes without crashes, hangs, or failure symptoms, but either the output file or the *stdout* is different than the output generated by the fault-free run. Most of our applications produce a single output file.

2.3. Use Cases

This section shows how SASSIFI can be employed to analyze resilience characteristics of applications. Specifically, we demonstrate how to configure SASSIFI to perform the following four studies.

1) What is the probability that a particle-strike on the register file while an application is running will produce an SDC? Answering this question allows us to quantify the benefit of adding ECC to the register file. If SDC probability is the key metric of interest and the value is low, the energy cost of enabling ECC may not be justified.
2) What is the probability that a bit-flip in the destination register of an executing instruction will result in an SDC? This study aims to quantify the effect of bit-flips in low-level unprotected state at the application-level by injecting the manifestation at the architecture-level, a commonly studied topic [6], [7], [8], [9].
3) What instruction types are likely to produce more SDCs when subjected to errors in destination registers? This study can provide insights into what to protect while employing selective instruction-level protection/duplication schemes.
4) How do SDC probabilities change when different architecture-level states are subjected to errors (e.g., injecting errors into values versus addresses of executing instructions)? How do the results change if we inject different bit-flip patterns (e.g., injecting single versus double bit-flips)? Addressing these questions provide insight into the accuracy of the commonly used error model (single-bit flips in values) in evaluating resilience of an application from bit-flips in low-level state (e.g., flip-flops) and what to consider for future studies.

For these studies, we must inject errors using different error models. Since SASSIFI injects errors through instrumentation handlers, we created three modes of operation which require instrumentation handlers to be inserted before, after, and both before and after the instruction, respectively.

We use the first mode to perform study (1) and call it the *RF mode* because it is used to measure register file AVF. We use the second mode to perform studies (2) and (3) by injecting errors into the destination register values by instrumenting instructions after they are executed. We call this mode *IOV* (Instruction Output Value). Finally, we use the third mode, which we call *IOA* (Instruction Output Address), to inject errors into destination register indices and store addresses. We use the IOA and IOV modes to perform the fourth study.

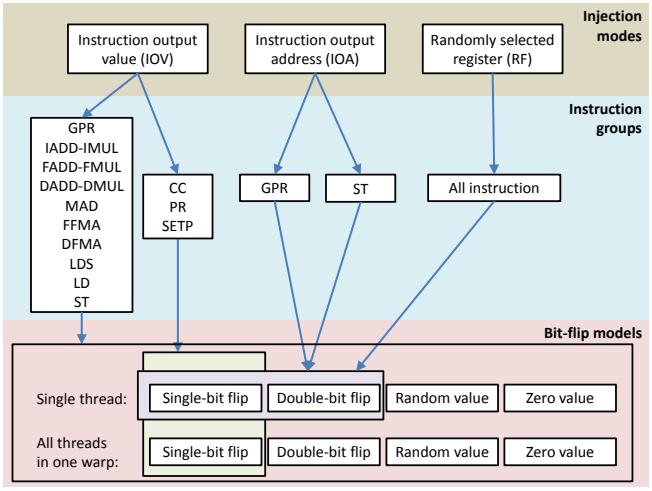

Figure 2: Summary of different error injection modes, instruction groups, and bit-flip models that SASSIFI provides.

Figure 2 depicts the different options in SASSIFI to select the instruction for injection and the bit-flip pattern to inject in these three modes below.

Figure 3 explains the SASSI handlers we wrote for the different modes.

Instruction selection and bit-flip models for the RF mode: This mode randomly selects a dynamic instruction from the program and injects an error in a randomly selected register among the allocated registers. During the second step of SASSIFI (as described in Section 2.2), we must specify *when* and *what* error to inject. For *when*, we randomly select the dynamic instruction from the entire application. For *what*, we generate and include two random numbers in the tuple to (1) select an instruction's output for injection and (2) generate the injected value based on the selected bit-flip model, respectively. In RF mode, we select a register among the statically allocated registers per thread for the selected kernel using the first random number. We obtain the number of allocated registers from the NVIDIA compiler using the *-Xptxas -v* option in the profiling phase. The second random number is either zero or one.

During the third step of SASSIFI, we instrument before all instructions. This handler checks whether the selected dynamic instruction is reached for the selected dynamic kernel. At the selected dynamic instruction, we use the register number (selected above) to check whether it is used as one of the source registers by the current instruction. If not, we monitor the subsequent instructions. If the selected register is written before being read, we mark the execution as masked and terminate early. If the selected register is among the sources in the current instruction, we inject the error. Since the goal is to represent direct particle strikes in

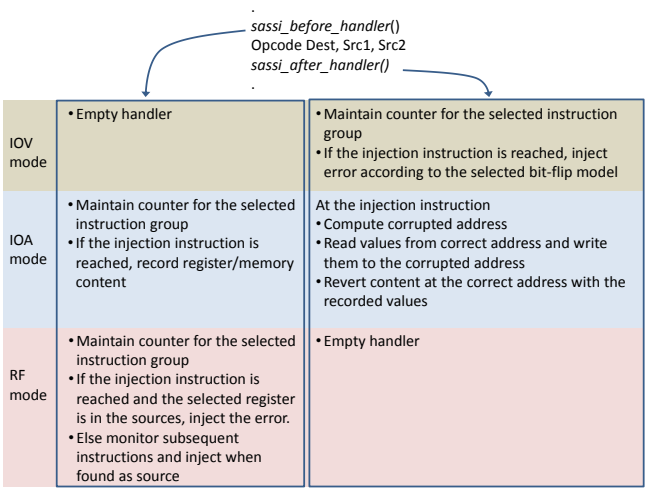

Figure 3: Error injection handlers for different modes.

the register file in this mode, we inject single and double bit-flips. Once the error is injected, the program resumes execution. If the thread (within the kernel) completes its execution and the selected register was never read, we mark the simulation as masked and terminate the simulation early.

Results obtained from these injections show the probability with which a particle strike in an allocated register manifests in the application output. These results must be further derated by the fraction of physical registers that are unallocated to obtain the AVF of the register file for a specific device.

We obtain the average number of allocated physical registers per kernel by multiplying the number of statically allocated registers per thread with the average number of active threads (number of active warps $\times$ 32) in an SM. We divide this value by the total number of physical registers in the SM to obtain the fraction of allocated registers. We obtain the average number of active warps from the *achieved_occupancy* metric as printed by the *nvprof* tool [10]. We obtain the number of allocated registers from the compilation step as described above. We use these per-kernel derating factors and weigh them with the relative kernel runtimes for each application to obtain a per-application derating factor.

Instruction selection and bit-flip models for the IOV mode: In the IOV mode, we randomly select an instruction for injection based on predefined instruction groups to study the sensitivity of error propagation. Based on the instruction types, we define the following instruction groups.

- Instructions that write to general-purpose registers (GPR)
- Instructions that write to condition-code registers (CC)
- Instructions that write to predicate registers (PR)
- Store instructions (ST)
- Integer add and multiply instructions (IADD-IMUL)
- Single-precision floating-point add and multiply instructions (FADD-FMUL)
- Double-precision floating-point add and multiply instructions (DADD-DMUL)
- Integer fused multiply and add instructions (MAD)
- Single-precision floating-point fused multiply-add instructions (FFMA)
- Double-precision floating-point fused multiply-add instructions (DFMA)
- Instructions that compare source registers and set a predicate register (SETP)
- Loads from shared memory (LDS)
- Load instructions, excluding LDS instructions (LD)

During the second step of SASSIFI, we specify *when* to inject errors by randomly selecting dynamic instructions from the entire application for the selected instruction group. For *what*, we pass a random number to select a destination register from the executing instruction for injection. Since soft errors in low-level state can manifest in the destination registers in many ways, we define the following bit-flip models (BFMs) for the IOV mode. We inject (1) single bit-flip, (2) double bit-flips, (3) random value, and (4) zero value in a single register/store value in one thread. We also define four more BFMs by injecting these four bit-flip patterns into the same register/store value in all of the active threads in a warp. We only inject single bit-flips in a single register in one thread for the CC and PR instruction groups. For the SETP instruction group, we inject using single bit-flip and warp-wide single bit-flip models.

During the third step of SASSIFI, we instrument the instructions after the instructions that write to a register or memory location. In the handler, we check whether the selected dynamic instruction for the selected instruction group is reached for the selected static and dynamic kernel invocation. At this instruction, we use the random number to select the destination register. For stores, we always inject the error in the value that is being stored.

Instruction selection and bit-flip models for the IOA mode: In this mode, we randomly select an instruction for injection based on two predefined instruction groups – GPR and ST. During the second step of SASSIFI, we specify *when* to inject the error by randomly selecting dynamic instructions from the entire application for the selected instruction group. For *what*, we pass a random number to select a destination register index from the executing instruction for injection. For ST instructions, we inject errors in the address.

During the third step of SASSIFI, we instrument the instructions before and after the instructions that write to a register or memory location. In the instrumentation handler that executes before the instruction, we record the register and memory content from the correct register index and address, respectively. In the instrumentation handler that executes after the instruction, we check whether the selected dynamic instruction for the selected instruction group is reached for the selected static and dynamic kernel invocation. At this instruction, we read the data from the correct register index or memory address and write it to the new error-injected (corrupted) register index or memory address. We then use the recorded data to revert the data in the correct location. In this mode, we inject single and double bit-flips; the bit position for flips is generated using the second random number, passed during the second step.

3. Evaluation

Our experimental flow targets NVIDIA Kepler and Maxwell architecture-based GPUs with Compute Capability 3.5 and 5.2 [11], [12]. Since we inject errors into the architecture state, our injection results do not depend on the specific GPU we use for experimentation as long as the binary file does not change. We used multiple systems with Tesla K20 and K40 GPUs, the CUDA 7.0 toolkit, and display driver versions 340.46 and 340.118 to obtain results for the Kepler GPUs. We obtain the results for the Maxwell GPUs using a Quadro M5000 board, the CUDA 7.0 toolkit, and display driver version 370.28.

We performed the profiling experiments to measure SASSIFI slowdowns on a system with an Intel i7-3930K CPU (3.2GHz), 32GB host memory, and a K40 GPU. We obtain the application-level runtime by taking the average runtime from three consecutive runs after a warm-up run. We obtain the GPU execution time using the *–print-gpu-trace* option in the *nvprof* tool, including the time spent in copying data between the GPU and CPU memories. Since CUPTI and *nvprof* cannot be used simultaneously, we obtain the GPU time for the SASSIFI runs by disabling CUPTI. We show these results in Section 3.5.

We use all applications from the Rodinia benchmark suite (version 2.3), which includes a diverse set of workloads from domains such as data mining, bioinformatics, medical imaging, image processing, physics simulation, and graph algorithms [5]. We also used two DOE mini-apps, CoMD and Lulesh, which are proxies of real applications run on supercomputers. CoMD is a reference implementation of classical molecular dynamics algorithms as used in materials science [3]. LULESH is a proxy application that represents typical workloads found in production Lagrangian hydrodynamics code [4]. We ensure deterministic outputs needed to detect SDCs by fixing the seed to the random number generator for the workloads that use variable seeds.

While we evaluated the resilience of all these workloads, we used the two DOE mini-apps to conduct detailed sensitivity analyses such as understanding the differences in injecting errors in different architecture state, using different bit-flip models, and quantifying ISA-level differences. In the rest of this section, we show the results for each of the use cases listed in Section 2.3.

3.1. Results for Use Case 1 (RF Mode)

We performed 500 injections each for CoMD and Lulesh using the single and double bit-flip models in the RF mode and show the results in Figure 4. The confidence interval for results based on 500 injections is at most 4.4% at the 95% confidence level.

To obtain the SM register file AVF, we derate the results with the average fraction of occupied physical registers. For CoMD and Lulesh (with our chosen inputs), these factors are 0.77 and 0.87, respectively. Based on these factors and SASSIFI SDC probabilities, we calculate the register file SDC AVF for CoMD and Lulesh as 7.5% and 0.7%, respectively, for the single-bit flip model. The respective numbers for the double-bit flip model are 8.8% and 0.5%.

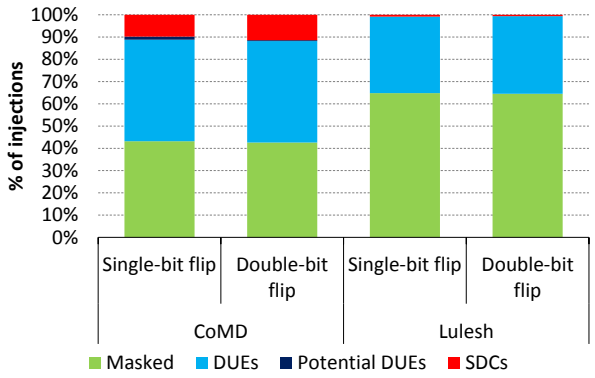

Figure 4: Outcomes of the injections into the allocated registers for CoMD and Lulesh in the RF mode.

These SDC AVF numbers allow us to quantify the benefit of adding parity/ECC to the register file. If the (derated) SDC probability is low for single and double bit-flips in an application then the energy cost of enabling parity/SECDED ECC may not be justified, as demonstrated by Lulesh. However, enabling parity or SECDED ECC seems important for CoMD, as a significant fraction of injections resulted in SDCs.

In this mode, we terminate the error simulations early if the injected error is never consumed. This optimization saved approximately 6% and 9% of the total runtime for CoMD and Lulesh, respectively.

3.2. Results for Use Case 2 (IOV Mode)

For this use case, we injected single-bit flips in the destination general-purpose registers of executing instructions. We injected 1000 errors each in all our workloads. These injection results have maximum error bars of 3% at the 95% confidence level. Figure 5 shows the observed masking, DUE, potential DUE, and SDC probabilities for all of the workloads. At least 60% of the injected errors did not have any effect on the program output, which is interesting given that we injected errors in the destination registers, not randomly selected registers (which may be completely dead at the point of injection). A significant fraction of the non-masked injection runs resulted in DUEs and potential DUEs. Results show that the SDC probabilities vary significantly based on the application, from 0% to 21%, which is expected since application-level error propagation highly depends on the data-flow of the workloads.

For applications with multiple static kernels that execute, identifying kernels that are more vulnerable to architecture-level errors provides insights into which kernels to protect for a cost-effective mitigation solution. Data obtained from this error injection campaign allows us to perform such analysis. Since we select instructions for error injection randomly across all dynamic instructions (except control flow instructions) in an application, the number of injections

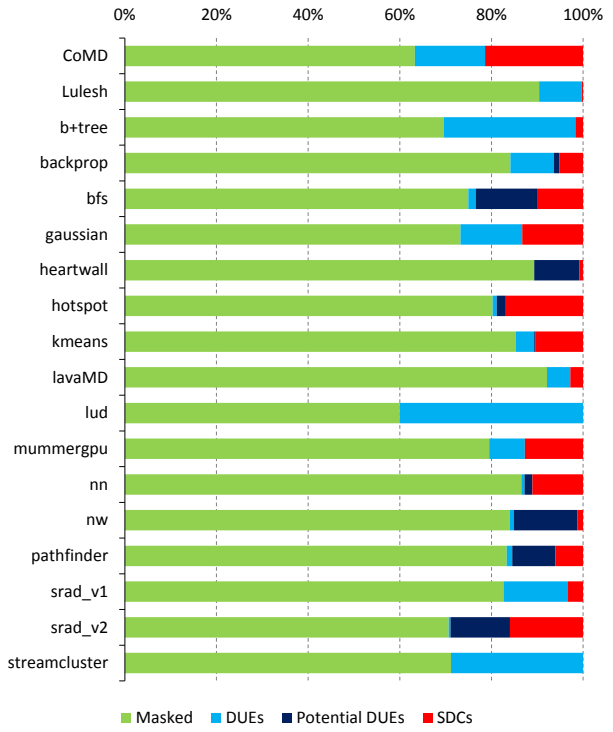

Figure 5: Outcomes of injecting single bit-flip errors in destination registers of instructions that write to general-purpose registers.

performed per static kernel can be approximated as the relative number of dynamic instructions per static kernel across all of its instantiations.

Figure 6 shows the per-kernel breakdown of the injection results obtained in Section 3.2 for three workloads. We do not plot the results from the static kernels where less than 10 injections were performed. The results show that most of the SDCs in the backprop application come from the bpnn_adjust_weights_cuda kernel. This kernel executes for only a third of the GPU execution time, excluding host execution and data transfer between the host and GPU. This result suggests that selective application-level error mitigation schemes such as full thread-level duplication can be cost-effective [13].

For kmeans, however, the most SDC susceptible kernel (invert_mapping) constitutes a significant fraction of the GPU execution time. For CoMD, we found that almost all kernels that execute for a significant fraction of time contribute to the SDC rate, which implies that intra-kernel level analysis should be performed to identify cost-effective ways to reduce SDCs.

3.3. Results for Use Case 3 (IOV Mode)

SASSIFI can be used to study the SDC sensitivity of different instruction groups that write to general-purpose registers to gain insights about what to protect in an application. As a case study, we injected 500 single bit-flip errors in each of the six different instruction groups

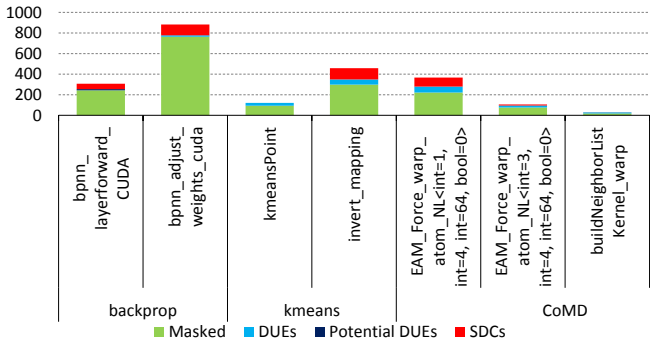

Figure 6: Breakdown of the error injection outcomes per static kernel of selected workloads. The height of each bar shows the number of injections performed per static kernel.

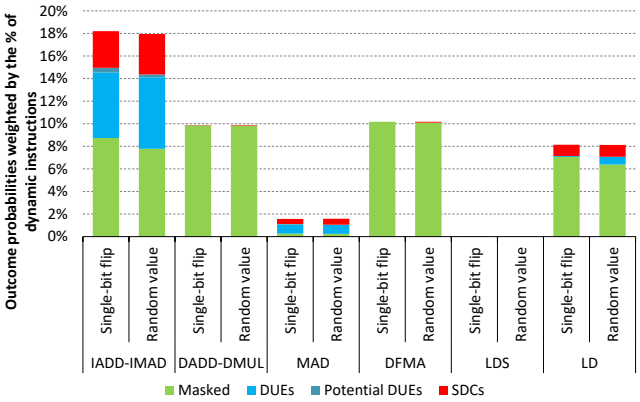

Figure 7: Outcomes of injections in different instruction groups for CoMD.

that write to general-purpose registers and have non-zero dynamic instruction counts. These results, when weighted with the total fraction of dynamic instructions in each group, can identify the instruction group that is responsible for most of the SDCs. Figure 7 shows the combined outcome probabilities weighted by the dynamic instruction fractions for CoMD.

These results show that errors in the DADD-DMUL and DFMA instruction groups are far less susceptible to producing SDCs than the other instruction groups. Although a significant fraction of the errors in LDS instructions result in SDCs, their contribution towards the total SDC rate is low because the number of dynamic LDS instructions is low for CoMD.

We also show the results obtained by injecting 500 random value errors for the six instruction groups in Figure 7 to illustrate the sensitivity of the results to the bit-flip model. These results indicate that for some instruction groups (IADD-IMUL, DADD-DMUL, MAD, and DFMA), the SDC or DUE probabilities do not change significantly. We notice a significant change in the SDC probability for LDS instructions and the DUE probabilities for LDS and LD instruction groups. This result indicates that using just the single bit-flip injection model might provide inaccurate

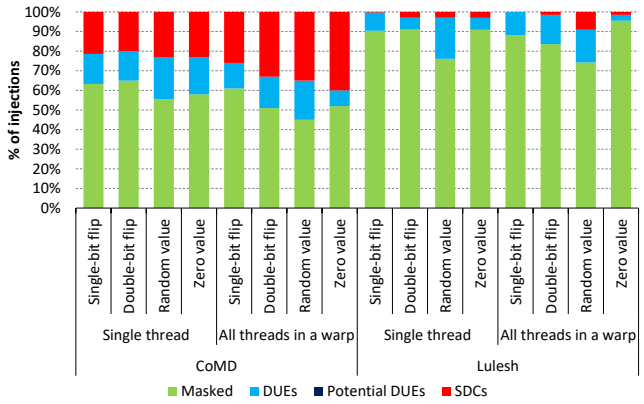

Figure 8: Outcomes of injections in GPR instructions using different bit-flip models for CoMD and Lulesh.

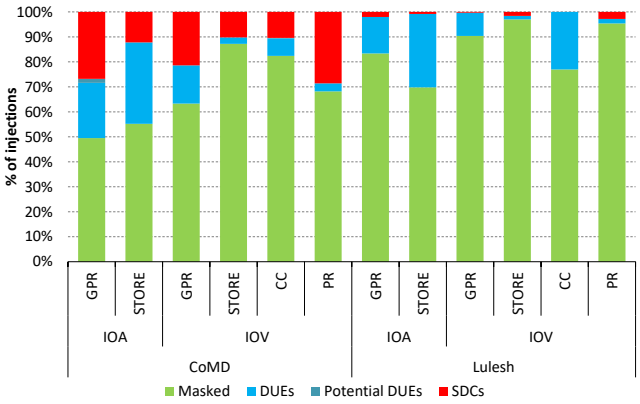

Figure 9: Outcome distribution from IOA and IOV mode injections in different architecture state for CoMD and Lulesh.

estimates of the actual error rates.

3.4. Results for Use Case 4 (IOV and IOA Modes)

To understand how the SDC probabilities change with different bit-flip models, we performed 500 injections for CoMD and Lulesh using the different bit-flip models described in Section 2.3 for the GPR instruction group. The results are shown in Figure 8.

Injecting more bit errors (e.g., random value error, warp-wide injections) are expected to result in either more DUEs or SDCs depending on how the errors propagate for each workload. From Figure 8, we observe that the SDC probability tends to increase with the number of bit-flips (single bit-flip to random value error), matching the expectation. We also observe that the warp-wide injections yield higher SDCs than the single thread injections for the same bit-flip pattern for CoMD. However, we see little difference between warp-wide and single-thread injections for Lulesh. We conducted similar injection experiments for all other supported instruction groups for CoMD and Lulesh and observed trends similar to the ones in Figure 8. Further analysis of the applications is needed to understand the reasons for the SDC increase.

We also obtain and analyze the results from injections in CC and PR register values, store values and addresses, and general-purpose register indices. Figure 9 shows the results obtained by injecting 500 errors per instruction group. Injecting single-bit flip errors in addresses (register index or store address) in GPR and ST instruction groups often results in higher SDC and DUE probabilities when compared to respective single-bit value errors.

For the IOA mode injections, we also injected 500 errors each for the double bit-flip pattern to study the sensitivity to the chosen bit-flip model. We observed only small changes in the SDC and DUE probabilities for these two workloads.

Injections into PR registers show higher SDC probabilities than CC registers, both of which are different than GPR value injections. Since the SDC probabilities from injections using different error models are different, it is important to quantify how the bit-flips in low-level unprotected state manifest at the architecture level for an accurate total application vulnerability analysis. Such an analysis requires a model that propagates microarchitecture state, such as a low-level simulator or an accelerated beam experiment on real hardware.

Since SASSIFI injects errors at the assembly language (SASS) level, it can be used to understand if an ISA change has a significant effect on the application resilience characteristic. We performed 1000 error injections in CoMD and Lulesh using a single-bit flip model and the GPR instruction group, running on Kepler and Maxwell architectures with ISA version 3.5 and 5.2, respectively. While we do not show the results here, we find little difference in SDC and DUE outcome probabilities between the two ISA versions.

3.5. SASSIFI Performance

Because the injection experiments require hundreds or thousands of runs per application, the slowdown of the application due to instrumentation can be important. We evaluated SASSIFI runtimes and observed 1.02× to 166× slowdowns at the application-level; at the kernel level the slowdowns were higher, ranging from 5.2× to 488×. The application level slowdowns are lower because the host-side execution time does not change with SASSI. We also noticed that the slowdowns in the IOA mode are higher than in the IOV or RF modes for several applications. This higher slowdown is likely because we instrument both before and after the instructions in the IOA mode (as summarized in Figure 3).

We measure the performance of SASSIFI in MWIPS (Million Warp-Instructions Per Second) and show the results in Table 2. We observed 13-254 MWIPS for our workloads. The expected MWIPS for low-level (microarchitecture or RTL level) simulators is order of magnitude lower. For example, single-threaded GPGPU-sim achieves up to 0.1 MWIPS [14]. Parallel GPU simulators can provide a speedup of roughly 4× using 6 threads [14], which is still significantly slower than SASSIFI. Verilog simulators are

TABLE 2: SASSIFI Performance in Million Warp-Instructions Per Second (MWIPS).

	CoMD	Lulesh	Rodinia		
			Geomean	Max	Min
IOV mode	81	81	57	233	13
IOA mode	55	46	64	254	13
RF mode	94	85	57	229	12
GPGPU-Sim	<0.1				

even slower, running at only 5–30 thousand instructions per second [15].

SASSI takes a static approach and instruments all static instructions of certain types. In our setup, we instrument all instructions which significantly affects performance. This overhead can be lowered by instrumenting only selected static instructions or kernels for a group of injection runs. A dynamic instrumentation approach that only instruments the instruction that is selected for injection can significantly lower the overheads.

4. Related Work

One prior study developed a CUDA-GDB based tool called GPU-Qin to inject architecture-level errors to study application resilience [7], [16] . Some of its capabilities are similar to SASSIFI, including injecting single-bit flips into destination registers of executing instructions. Any code used to identify an injection site, inject the error, or monitor error propagation will execute on the host CPU for a CUDA-GDB based tool, which causes significant performance degradations. As a result, GPU-Qin requires complex steps during grouping and profiling phases to manage performance overhead. The instrumentation code that performs these functions executes on the GPU for SASSIFI.

The GPU-Qin paper did not study the effect of (1) faults in predicate registers, condition codes, and memory values, (2) injecting different bit-flip patterns (e.g., double bit flips, random value errors in destination registers), and (3) errors in different instruction types (e.g., floating-point versus integer instructions) to identify which instructions are likely to produce more SDCs. SASSIFI natively includes these capabilities to provide further insight into the development cost-effective error mitigation schemes.

Researchers have also proposed reliability evaluation tools at the LLVM IR level through injection [8] and PTX level program analysis [17], [18], which can be faster than SASSIFI. Techniques that operate at higher-levels such as C-code, LLVM IR, or PTX cannot accurately model errors in the native GPU ISA because SASS code-generation, including register allocation and ISA specific optimizations, are performed in the back-end compiler. Figure 10 shows a brief overview of where these levels lie in the compilation process. In fact, researchers have recently shown that the PTX based reliability measurements induce some underestimation of the actual hardware vulnerability [19]. They injected register file errors at the PTX and SASS levels and found the masking rate to be always higher at the PTX level.

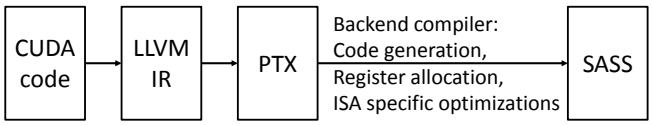

Figure 10: Brief overview of where different levels lie in the compilation process.

Since we study the vulnerability of the software through architectural-level error injections, our approach (especially in the IOV-mode) is similar to measuring the Program Vulnerability Factor as defined in [20]. We use error injection instead of performing ACE analysis [21], [22]. While injections from the RF mode can be used to directly calculate the AVF of the register file, injection from the IOV and IOA modes do not directly translate into AVF. The latter modes show how different architecture-level errors affect program outcomes. A full evaluation of AVF requires the distribution of error manifestations at the architecture level based on how faults propagate through the microarchitecture.

Some studies investigate GPU vulnerability using ACE analysis [23], [24] or fault injection [25], [19], but few study the effect of error propagation and masking all the way to the application output. For example, the GUFI framework [19] uses a publicly available GPU microarchitecture simulator to inject errors in the microarchitecture units to measure AVF. It can also perform injections at the SASS level, which are similar to SASSIFI injections, but needs a much slower microarchitecture simulator.

Fault injection can be performed at the RTL or gate level [26], [27], [28], [29] to understand the effect of errors on combinational and sequential logic and the resulting manifestation of these lower-level errors at the architecture level. Since SASSIFI uses an architecture-level error model, we are not able to capture the exact effect of lower-level errors (also observed by [28], [30], [29]). Instead, SASSIFI provides several architecture-level error models for sensitivity analysis. SASSIFI is significantly faster than lower-level error injection tools, which allows SASSIFI to not only inject more errors but also execute of full applications.

5. Summary and Future Work

This paper describes the SASSIFI tool that performs architecture-level fault injections to analyze GPU application resilience. SASSIFI operates in three steps: (1) profiling and identifying the error injection space; (2) statistically selecting error injection sites; and (3) injecting errors into executing applications and monitoring error behavior. Steps 1 and 3 use an assembly language level instrumentation tool called SASSI. SASSIFI provides the ability to perform a wide range of application resilience studies. Some examples include (1) quantifying the importance of enabling ECC for the register file for specific applications, (2) understanding which applications and GPU kernels are more susceptible to SDCs when subjected to architecture-level errors, (3) identifying the instruction groups that are likely to produce more SDCs when subjected to errors in destination registers,

and (4) measuring the application output level effect of corrupting different architecture states.

Some of the other interesting analyses that can be performed using SASSIFI include (1) extracting application characteristics that correlate with SDCs, which are key for developing cost-effective error mitigation schemes, (2) tracking inter-kernel error propagation patterns to identify where to place a cost-effective set of error detectors, and (3) analyzing why certain instruction groups are highly susceptible to SDCs when subjected to errors.

Soft-errors in low-level unprotected state can manifest at the architecture-level in several ways. While SASSIFI provides the capability to inject such manifestations, identifying the rate of propagation for different architecture-level manifestations for a specific device of interest is needed to derive realistic application SDC rates.

References

[1] M. Stephenson, S. K. S. Hari, Y. Lee, E. Ebrahimi, D. R. Johnson, D. Nellans, M. O'Connor, and S. W. Keckler, "Flexible Software Profiling of GPU Architectures," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2015.

[2] NVIDIA, "CUPTI :: CUDA Toolkit Documentation," http://docs.nvidia.com/cuda/cupti/index.html, 2014.

[3] Exascale Co-Design Center for Materials in Extreme Environments (ExMatEX), "CoMD: Classical Molecular Dynamics Proxy Application," http://exmatex.github.io/CoMD.

[4] Lawrence Livermore National Laboratory, "Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH) 2.0," http://codesign.llnl.gov/lulesh.php, 2013.

[5] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, 2009.

[6] S. K. S. Hari, S. V. Adve, H. Naeimi, and P. Ramachandran, "Relyzer: Exploiting Application-level Fault Equivalence to Analyze Application Resiliency to Transient Faults," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2012.

[7] B. Fang, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi, "GPU-Qin: A Methodology for Evaluating the Error Resilience of GPGPU Applications," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014.

[8] G. Li, K. Pattabiraman, C.-Y. Cher, and P. Bose, "Understanding Error Propagation in GPGPU Applications," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2016.

[9] J. Wei, A. Thomas, G. Li, and K. Pattabiraman, "Quantifying the Accuracy of High-Level Fault Injection Techniques for Hardware Faults," in *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, June 2014.

[10] NVIDIA, "Profiler Users's Guide," http://docs.nvidia.com/cuda/profiler-users-guide, September 2015.

[11] ——, "NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110," https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf, 2012.

[12] ——, "NVIDIA GeForce GTX 980 Featuring Maxwell, The Most Advanced GPU Ever Made." http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF, 2014.

[13] J. Wadden, A. Lyashevsky, S. Gurumurthi, V. Sridharan, and K. Skadron, "Real-World Design and Evaluation of Compiler-Managed GPU Redundant Multithreading," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2014.

[14] S. Lee and W. W. Ro, "Parallel GPU Architecture Simulation Framework Exploiting Work Allocation Unit Parallelism," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2013.

[15] C. Celio, D. A. Patterson, and K. Asanovic, "The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor," in *Technical Report No. UCB/EECS-2015-167*, 2015.

[16] NVIDIA, "CUDA-GDB :: CUDA Toolkit Documentation," http://docs.nvidia.com/cuda/cuda-gdb/index.html, 2014.

[17] ——, "PTX ISA :: CUDA Toolkit Documentation," http://docs.nvidia.com/cuda/parallel-thread-execution/, September 2015.

[18] S. Li, V. Sridharan, S. Gurumurthi, and S. Yalamanchili, "Software-based Dynamic Reliability Management for GPU Applications," in *International Reliability Physics Symposium*, 2016.

[19] S. Tselonis and D. Gizopoulos, "GUFI: A Framework for GPUs Reliability Assessment," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2016.

[20] V. Sridharan and D. Kaeli, "Eliminating Microarchitectural Dependency from Architectural Vulnerability," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2009.

[21] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, "A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, December 2003.

[22] A. Biswas, P. Racunas, R. Cheveresan, J. Emer, S. S. Mukherjee, and R. Rangan, "Computing Architectural Vulnerability Factors for Address-Based Structures," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2005.

[23] J. Tan, N. Goswami, T. Li, and X. Fu, "Analyzing Soft-error Vulnerability on GPGPU Microarchitecture," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, 2011.

[24] H. Jeon, M. Wilkening, V. Sridharan, S. Gurumurthi, and G. H. Loh, "Architectural Vulnerability Modeling and Analysis of Integrated Graphics Processors," in *Workshop on Silicon Errors in Logic - System Effects (SELSE)*, 2013.

[25] N. Farazmand, R. Ubal, and D. Kaeli, "Statistical Fault Injection-Based AVF Analysis of a GPU Architecure," in *Workshop on Silicon Errors in Logic - System Effects (SELSE)*, 2012.

[26] G. P. Saggese, N. J. Wang, Z. T. Kalbarczyk, S. J. Patel, and R. K. Iyer, "An Experimental Study of Soft Errors in Microprocessors," *IEEE Micro*, vol. 25, no. 6, pp. 30–39, 2005.

[27] M. Maniatakos, N. Karimi, C. Tirumurti, A. Jas, and Y. Makris, "Instruction-Level Impact Analysis of Low-Level Faults in a Modern Microprocessor Controller," *IEEE Transactions on Computers*, vol. 60, no. 9, pp. 1260–1273, 2011.

[28] H. Cho, S. Mirkhani, C.-Y. Cher, J. A. Abraham, and S. Mitra, "Quantitative Evaluation of Soft Error Injection Techniques for Robust System Design," in *Design Automation Conference (DAC)*, 2013.

[29] E. Cheng, S. Mirkhani, L. G. Szafaryn, C. Y. Cher, H. Cho, K. Skadron, M. Stan, K. Lilja, J. Abraham, P. Bose, and S. Mitra, "CLEAR: Combining Hardware and Software Techniques to Tolerate Soft Errors in Processor Cores," in *Design Automation Conference (DAC)*, 2016.

[30] M. Gschwind, V. Salapura, C. Trammell, and S. A. McKee, "SoftBeam: Precise Tracking of Transient Faults and Vulnerability Analysis at Processor Design Time," in *Proceedings of the International Conference on Computer Design (ICCD)*, 2011.