SASSIFI User Guide

Siva Kumar Sastry Hari (shari@nvidia.com)

March 20, 2017

Contents

1 Introduction

SASSIFI provides an automated framework to perform error injection campaigns for GPU application resilience evaluation. SASSIFI builds on top of SASSI, which is a low-level assembly-language instrumentation tool that provides the ability to instrument instructions in the low-level GPU assembly language (SASS) [1].

SASSIFI can be used to perform many types of resilience studies. Some examples are:

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 the SDC probability is low, the energy cost of enabling ECC may not be justified. This is a commonly used architecture-level error model.

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 in academia.

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 vs. addresses of executing instruction)? How do the results change if we inject different bit-flip patterns (e.g., injecting single vs. 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 want to 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.

For studies such as (1), we designed a mode called the *RF mode* where we inject bit-flips in the Register File (RF), randomly spread across time and space (among allocated registers). For studies such as (2) and (3), we designed a mode that injects errors into the destination register values by instrumenting instructions after they are executed. We call this mode *IOV* (Instruction Output Value). Finally, we desinged the third mode called the *IOA* (Instruction Output Address) mode to inject errors into destination register indices and store addresses. We can use the IOA and IOV modes to perform the fourth study.

For more details about how SASSIFI can be used to conduct resilience evaluation studies and some results, please see the SASSIFI ISPASS 2017 paper [2]. The resilience case study in the SASSI ISCA 2015 paper [1] and SASSIFI SELSE 2015 presentation [3] also offer some details and results. Portions of the documentation are from our IPASS 2017 paper, which is copyrighted by IEEE.

2 Where can SASSIFI inject errors?

For the IOA-mode (instruction output-level injections), SASSIFI can inject errors in the outputs of randomly selected instructions. SASSIFI allows us to select different types of instructions to study how error in them will propagate to the application output. As of now (3/10/2017), SASSIFI supports selecting the following instruction groups.

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

SASSIFI can be extended to include custom instruction groups. Follow instructions in Section 6 to create new instruction groups. Details about the current instruction grouping, i.e., which SASS instructions are included in different groups, can be found in $SASSIFI_HOME/err_injector/error_injector.h.

In the IOA mode, SASSIFI supports selecting the following two instruction groups – GPR and ST. At these instructions, SASSIFI injects errors into the address (register index or store address) based on the following defined bit-flip models.

For the RF mode (injections to measure RF AVF), SASSIFI selects a dynamic instruction randomly from a program and injects an error in a randomly selected register among the allocated registers. Results

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

We can 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 with the total number of physical registers in the SM to obtain the fraction of allocated registers. We can obtain the average number of active warps from the *achieved_occupancy* metric as printed by the *nvprof* tool. We can obtain the number of allocated registers during the compilation step using *-Xptxas -v* option. We then use these per kernel dertaing factors and weigh them with the relative kernel runtimes for each application to obtain a per application derating factor.

3 What errors can SASSIFI inject?

For the IOV mode, SASSIFI can inject the error in a destination register based on the different Bit Flip Models (*BFM*). In the current release, the following BFMs are implemented.

1. Single bit-flip: one bit-flip in one register in one thread
2. Double bit-flip: bit-flips in two adjacent bits in one register in one thread
3. Random value: random value in one register in one thread
4. Zero value: zero out the value of one register in one thread
5. Warp wide single bit-flip: one bit-flip in one register in all the threads in a warp
6. Warp wide double bit-flip: bit-flips in two adjacent bits in one register in all the threads in a warp
7. Warp wide random value: random value in one register in all the threads in a warp
8. Warp wide zero value: zero out the value of one register in all the threads in a warp

In the current implementation, we can only inject single bit-flip in one register in one thread (first bit-flip model) for the CC and PR injections. For the SETP-OP instruction group, we can inject only single bit-flip and warp wide single bit-flip (first and fifth bit-flip models, respectively).

For the IOA and RF modes, SASSIFI considers the following two bit-flip models.

- Single bit-flip
- Double bit-flip

These BFMs can be extended to include different bit-flip pattern. To add a new bit-flip model err_injector.h and injector.cu files in err_injector directory and common_params.py and specific_params.py files in the scripts directory need to be modified.

4 Getting started with SASSIFI

4.1 Prerequisites

- A linux-based system with an x86 64-bit host, a Fermi-, Kepler-, or Maxwell-based GPU. SASSIFI has been tested on Ubuntu (12) and CentOS (6).
- Python 2.7 is needed to run the scripts provided to generate injection sites, launch injection campaign, and parse the results.
  - The lockfile module is needed to run the injection jobs in parallel either on a multi-gpu system or a cluster of nodes with shared filesystem. Instructions to install the lockfile module can be found here [4]. This module is not needed if you want to run injection jobs sequentially on a single-gpu system.

  - (Optional) The final results can be parsed into an xlsx file using the xlsxwriter module. Instructions to install xlswriter can be found here [5]. The results will be parsed into multiple text files, if you do not have xlswriter. These can be copied into an excel file to plot and visualize the results.

- SASSI, which can be downloaded from GitHub [6]. SASSIFI is tested using the latest commit (5523d984ad047a272297c1a3ff8c63f55c0ad026). SASSIFI provides code that needs to be compiled using the SASSI framework. This code includes SASSI handlers that execute code before and after instructions for profiling and error injections. Please follow the steps provided in the SASSI documentation to install SASSI.

4.2 SASSIFI package directory structure

```
SASSIFI_HOME
├── err_injector ...  Source code of the SASSI handlers for application profiling and
│                     error injection. This directory should be moved to the SASSI
│                     source directory.
│   ├── error_injector.h
│   ├── injector.cu
│   ├── profiler.cu
│   ├── Makefile
│   └── copy_handler.sh
├── scripts ...  Scripts to generate injection list, run injections, and parse
│                results.
│   ├── common_params.py
│   ├── specific_params.py
│   ├── common_functions.py
│   ├── generate_injection_list.py
│   ├── run_one_injection.py
│   ├── run_injections.py
│   ├── parse_results.py
│   └── process_kernel_regcount.py
├── suites ...  Workloads will be stored here.
│   └── example ...  We provide a sample benchmark suite named example.
│       └── simple_add ...  Look at the makefile here.
│           ├── simple_add.cu
│           └── Makefile
├── run ...  Stores run and sdc_check scripts for different applications. The
│            subdirectory structure here is similar to the suites directory
│            above.
│   └── example
│       └── simple_add
│           ├── sassifi_run.sh
│           └── sdc_check.sh
├── bin ...  Stores application binaries. This directory will be auto
│            generated by the makefile in the suites subdirectory.
│   ├── none ...  Application binaries without instrumentation.
│   ├── profiler ...  Application binaries instrumented for application profiling.
│   ├── inst_injector ...  Application binaries instrumented with instruction output-level
│   │                      error injection handler.
│   └── rf_injector ...  Application binaries instrumented with register file level error
│                        injection handler.
└── logs ...  Logs will be stored here after error injection runs. This
              directory will be auto generated by the scripts.
    └── example ...  One directory per benchmark suite will be created.
```

```
    |   |   |-- simple_add ...  One directory per workload. This directory will contain files
    |   |                       that list the error injection sites and the results from the
    |   |                       error injection campaigns.
    |   |-- results ...  Summary of the parsed results (excel sheet) will be stored here.
    |-- test.sh ...  Sample script that automates several steps involved running
    |                SASSIFI. This is a great place to start, if you are short on
    |                time.
    |-- docs
        |-- sassifi-user-guide.pdf ...  This file!
        |-- tex ...  LaTeX files used to generate this document.
```

4.3 Setting up and running SASSIFI

Follow these steps to setup and run SASSIFI. We provide a sample script (test.sh) that automates several of these steps.

1. **Set the following environment variables**:
   - SASSIFI_HOME: Path to the SASSIFI package (e.g., /home/username/sassifi_package/)
   - SASSI_SRC: Path to the SASSI source package (e.g., /home/username/sassi/)
   - INST_LIB_DIR: Path to the SASSI libraries (e.g., SASSI_SRC/instlibs/lib/)
   - CCDIR: Path to the gcc version 4.8.4 or newer (e.g., /usr/local/gcc-4.8.4/)
   - CUDA_BASE_DIR: Path to SASSI installation (e.g., /usr/local/sassi7/)
   - LD_LIBRARY_PATH should include the cuda libraries (e.g., CUDA_BASE_DIR/lib64/ and CUDA_BASE_DIR/extras/CUPTI/lib64/)
   - Ensure that the GENCODE variable is correctly set for the target GPU in SASSI_SRC/instlibs/env.mk and application makefiles (e.g., SASSIFI_HOME/suites/example/simple_add/Makefile.

2. **Copy the SASSI Fault Injection (SASSIFI) handler into the SASSI package**: We provide err_injector/copy_handler.sh script to perform this step. Simply run it from any directory. This script creates a new directory named err_injector in the SASSI_SRC/instlibs/src directory and creates soft-links for the files provided in the err_injector directory to avoid keeping multiple copies of the SASSI handler files.

3. **Compile the SASSIFI handlers**: Simply type *make* in $SASSI_SRC/instlibs/src/err_injector. This should create four libraries. The first one is for profiling the application and identifying how many injection points exist. The remaining three are for injecting errors during an application run (one each for the three injection modes).

4. **Prepare applications**:
   (a) **Record fault-free outputs:** Record golden output file (as golden.txt) and golden stdout (as golden_stdout.txt) and golden stderr (as golden_stderr.txt) in the workload directory (e.g., $SASSIFI_HOME/suites/example/simple_add/).
   (b) **Create application-specific scripts:** Create sassifi_run.sh and sdc_check.sh scripts in run/ directory. These are workload specific and have to be manually created. Instead of using absolute paths, please use environment variables for paths such as BIN_DIR, APP_DIR, DATA_SET_DIR, and RUN_SCRIPT_DIR. These variables are set by run_one_injection.py script before launching error injections. See the bash scripts in the run/example/simple_add/run/ directory for examples. You can also add an application specific check here.
   (c) **Prepare applications to compile with the SASSIFI handlers**: This might require some work. Follow instructions in the SASSI documentation on how to compile your application with a SASSI handler.

Tip: Prepare them such that you can type "make OPTION=profiler" to generate binaries to do the profiling step (step 4) and "make OPTION=inst_value_injector" or "make OPTION=inst_address_injector" or "make OPTION=rf_injector" to generate binaries for error injection campaigns for the three injection modes (see Sections 1 and 2). See makefile in suites/example/simple_add/ for an example. This makefile installs different versions of the binaries to $SASSIFI_HOME/bin/$OPTIONS/ directories.

5. **Profile the application**: Compile the application with "OPTION=profiler" and run it once with the same inputs that is specified in the sassifi_run.sh script. A new file named sassifi-inst-counts.txt will be generated in the directory where the application was run. This file contains the instruction counts for all the instruction groups defined in err_injector/error_injector.h and all the opcodes defined in sassi-opcodes.h for all the CUDA kernels. One line is created per dynamic kernel invocation and the format in which the data is printed is shown in the first line in the sassifi-inst-counts.txt file.

6. **Build the applications for error injection runs**: Simply run "make OPTION=inst_value_injector", "make OPTION=inst_address_injector" and/or "make OPTION=rf_injector"

7. **Generation injection sites**:

   (a) Ensure that the parameters are set correctly in specific_params.py and common_params.py files. Some of the parameters that need user attention are:

   - Setting maximum number of error injections to perform per instruction group and bit-flip model combination. See NUM_INJECTION and THRESHOLD_JOBS in specific_params.py file.
   - Selecting instruction groups and bit-flip models. See rf_bfm_list, inst_value_igid_bfm_map, and inst_address_igid_bfm_map in specific_params.py for the list of supported instruction groups (IGIDs) and bit-flip models (BFMs). Simply uncomment the lines to include the IGID and the associated BFMs. User can also select only a subset of the supported BFMs per IGID for targeted error injection studies.
   - Listing the applications, benchmark suite name, application binary file name, and the expected runtime on the system where the injection job will be run. See the apps dictionary in specific_params.py for an example. The dictionary and the strings defined here are used by other scripts to identify the directory structure in the suites directory and the application binary name. The expected runtime defined here is used later to determine when to timeout injection runs (based on the TIMEOUT_THRESHOLD defined in common_params.py).
   - Setting paths for the suites, logs, bin, and run directories if the user decides to use a different directory structure. If the directory structure for the new benchmark suite that you plan to use is different, please update the app_dir[app] and app_data_dir[app] variables accordingly.
   - Setting the number of allocated registers per static kernel per application. When an application is compiled using *-Xptxas -v* flags, the number of registers allocated for each static kernel in the application are printed on the standard error (stderr). User needs to parse the stderr and update the num_regs dictionary in the specific_params.py file. Obtain the number of allocated registers without SASSI instrumentation. If num_regs dictionary is incorrect (missing/extra kernel names, fewer/more registers per kernel), then the results will also be incorrect because the number of error injections are chosen based on num_regs. We provide the process_kernel_regcount.py script that parses the stderr from an input file and creates a dictionary per application which is stored in a pickle file. This pickle file can be loaded directly by the specific_params.py (see set_num_regs() for an example). We process the stderr generated by compiling the simple_add program using this script in test.sh.
     The num_regs dictionary is needed for the RF mode injections. If you do not plan to perform RF mode injections, you can ignore this part.

   (b) Run generate_injection_list.py script to generate a file that contains what errors to inject. Instructions are selected randomly for across the entire application for the RF mode and across the instructions from the specified instruction group in the IOV and IOA modes. Since we know

the instruction count breakdown per kernel invocation from the profiling phase, we combine the instructions from all the kernel executions (based on the instruction groups) and randomly select dynamic instruction numbers for error injections. We map this dynamic instruction number back to a dynamic kernel invocation index, along with static kernel name. We create a random number (between 0 and 1) for selecting the destination register among the number of destination registers in the selected dynamic instruction for the IOV and IOA modes. We select the register number within the set of allocated registers for the selected static kernel for the RF mode. We also select an additional random number for selecting the bit location to inject the error (according to the chosen bit-flip model).

8. **Run injections**: Run the run_injections.py script to launch the error injection campaign. This script will run one injection after the other in the standalone mode. Please do not attempt to run multiple jobs in parallel unless you install the lockfile python module or modify the run_one_injection.py script such that it does not write to the same results file. If you use the multigpu option, mutiple injection jobs will be launched in parallel depending on the number of GPUs present in the system and the parameter specified in specific_params.py (NUM_GPUS). If you have a cluster of nodes where you can launch injection jobs, you can write some code in the "check_and_submit_cluster" function in run_injections.py script to launch multiple jobs to the cluster.

   Tip: Perform a few dummy injections before proceeding with full injection campaign. Go to step 3 and look for DUMMY_INJECTION flag in the makefile. Setting this flag will allow you to go through most of the SASSI handler code, but skip the error injection. This is to ensure that you are not seeing crashes/SDCs that you should not see.

9. **Parse results**: Use the sample parse_results.py script to get an initial set of parsed results. This script generates an excel workbook with three sheets, if the xlsxwriter python module is found in the system. If not, three text files are created. The first sheet/text file shows the fraction of executed instructions for different instruction groups and opcodes. The second sheet/text file shows the outcomes of the error injections. Table 1 explains how we categorize error outcomes. The third sheet/text file shows the average runtime for the injection runs for different applications, instruction groups, and bit-flip models. Based on how you want to visualize the results, you may want to modify the script or write your own.

In the current setup, steps 1, 2, 3, 4b, 4c, and 7a have to be done manually. Once this is done, the remaining steps can be automated and we provide an example script (test.sh) to run these steps using a single command (./test.sh from $SASSIFI_HOME).

5 Error outcomes

Table 1 shows how we categorize the outcomes of the error injection runs.

6 Adding a new instruction group for error injections

As mentioned in Section 2, SASSIFI can be extended to include custom instruction groups. Here we outline the changes needed to add a new instruction group to SASSIFI.

- Assign a name to the new instruction group (e.g., NEW_OP) and add it to the *enum INST_TYPE* in err_injector/error_injector.h. Include it in the instCatName array in the same file.
- Identify the SASSI opcodes that should be included in this new group and update the get_op_category function in err_injector/error_injector.h accordingly. The list of available opcodes can be found in the sassi-opcodes.h.
- Specify what to do in the sassi_after_handler for error injection. For the IOV mode injections, simply add a *case* in the *switch* statement in the sassi_after_handler function in err_injector/injector.cu, similar

Table 1: Error injection outcomes.

Category	Subcategory	Explanation
Masked	Application output is same as the error free output. No error symptom is observed.	
	Value not read	Currently, this applies only to the RF mode injections. The register selected for error injection, but it was never read.
	Written before being read	Currently, this applies only to the RF mode injections. The register selected for error injection was overwritten before being read.
	Other reasons	For the RF mode injection, the injected error was consumed but masked later in the application. For the instruction output-level injections (IOV and IOA modes), this is the only the masked outcome subcategory.
DUE	Executions that terminate early or hang.	
	Timeout	Executions that do not terminate within an allocated threshold, which is configurable by changing TIMEOUT_THRESHOLD in scripts/common_params.py. Default is 10× the fault-free runtime.
	Non zero exit status	Application exits with non-zero exit status.
Potential DUE	Symptoms of an unsuccessful application run can be seen in either *stdout*, *stderr*, or kernel exit status. Executions with failure symptoms can be categorized as DUEs if the system has appropriate error monitors.	
	Kernel error, but masked	One of the kernels did not complete successfully (detected by comparing kernel exit status with *cudaSuccess*). The output of the application, however, matches the fault-free output.
	Kernel error, but SDC	One of the kernels did not complete successfully (detected by comparing kernel exit status with *cudaSuccess*). The output of the application does not match the fault-free output.
	Recorded error messages in *stderr*	Error messages are recorded in the *stderr*. For applications that write to *stderr* in fault-free runs, the new *stderr* is different than the fault-free one.
	Recorded error messages in *stdout*	Error messages are recorded in the *stdout*.
	dmesg error and *stderr* file is different	Stderr is different, but messages are recorded in the linux kernel (accessed using *dmesg* utility).
	dmesg error and stout file is different	*Stdout* is different, but messages are recorded in the linux kernel (accessed using *dmesg* utility).
	dmesg error and the output file is different	Output file (if it exists for the application) is different, but messages are recorded in the linux kernel (accessed using *dmesg* utility).
	dmesg error and application specific check failed	User-specified application specific (SDC) check failed, but messages are recorded in the linux kernel (accessed using *dmesg* utility).
SDC	Application finishes without crashes, hangs, or failure symptoms but at least one of the outputs of the application is different.	
	Stdout is different	Text printed in *stdout* is different. Output file generated by the application is identical to the fault-free run.
	Output is different	The output file generated by the application is different than the output generated by the fault-free run.
	Application-specific check failed	The application-specific check provided by the user failed.

to the IADD_IMUL_OP. For the IOA mode injections, add an *if* statement in the sassi_after_handler function similar to the STORE_OP instructions.

- Update the scripts such that error injection sites will be created and injection jobs will be launched for the new instruction group. Update the categories of the instruction types in scripts/common_params.py such that it matches the *enum INST_TYPE* in err_injector/error_injector.h. Finally add the new instruction group and the associated bit-flip models in the inst_value_igid_bfm_map or inst_address_igid_bfm_map in scripts/specific_params.py.

7 Bug reports

We plan to track issues using GitHub's issue tracking features.

8 Abbreviations

This document and the SASSIFI source code uses many abbreviations and we list important ones here:
SASSIFI: SASSI-based Fault Injector
RF: Register File
AVF: Architecture Vulnerability Factor
RF mode: Injection mode in which the register selected for injection is independent of the instruction executing at the time of injection. This mode is used to analyze RF AVF.
IOV mode: Injection mode in which the value selected for injection is dependent on the instruction executing at the time of injection. We inject errors in the output value of the instruction that just executed. This mode allows us to perform targeted error injections on various instruction groups.
IOA mode: Injection mode in which the address selected for injection is dependent on the instruction executing at the time of injection. We inject errors either in the register index or store address of the instruction that just executed, based on the instruction type. This mode allows us to perform targeted error injections to study the sensitivity of address errors.
SDC: Silent Data Corruption
DUE: Detected Uncorrectable Error
Pot DUE: Potential DUE (could be detected if proper checkers are in place)
BFM: Bit-Flip Model
IGID: Instruction Group ID
GPR: General Purpose Register
CC: Condition Code register
PR: Predicate Register

References

[1] Mark Stephenson, Siva Kumar Sastry Hari, Yunsup Lee, Eiman Ebrahimi, Daniel R. Johnson, Mike O'Connor, David Nellans, and Stephen W. Keckler. Flexible Software Profiling of GPU Architectures. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2015.

[2] Siva Kumar Sastry Hari, Timothy Tsai, Mark Stephenson, Stephen W. Keckler, and Joel Emer. Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS). 2017.

[3] Siva Kumar Sastry Hari, Timothy Tsai, Mark Stephenson, Joel Emer, and Stephen W. Keckler. Workshop on Silicon Errors in Logic - System Effects (SELSE). `http://www.selse.org/images/selse_2015/presentations/Hari.pdf`, 2015.

[4] lockfile 0.12.2 : Python Package Index. `https://pypi.python.org/pypi/lockfile`.

[5] Getting Started with XlsxWriter - XlsxWriter Documentation. `http://xlsxwriter.readthedocs.io/getting_started.html`.

[6] NVIDIA. SASSI: Flexible GPGPU instrumentation. https://github.com/NVlabs/SASSI, 2015.