SASSIFI User Guide

Siva Kumar Sastry Hari (shari@nvidia.com)

July 28, 2016

Contents

1 Introduction

SASSIFI can quantify how architecture-level errors propagate to the program output level through architecture-level error injections experiments. It can be used in two modes: (1) Inject errors in the outputs of the instructions. This is useful if you want to nd out what can happen if a low-level soft-error manifests at the architecture-level. See the Resilience case study in the SASSI ISCA 2015 paper [1] and SASSIFI SELSE 2015 presentation [2] for more details. We call this mode of operation as *D-mode* because the injections are dependent on the instruction. (2) Inject bit- ips in the Register File (RF), randomly spread across time and space (among allocated registers). The purpose of such injections is to compute Architectural Vulnerability Factor (AVF) of the RF. The results would tell us the importance of using ECC/parity on the RF. We call this mode *I-mode* because the register selected for injection is independent of the instruction executing at the time of injection.

2 Where can SASSIFI inject errors?

For the D-mode (instruction output-level injections), SASSIFI can inject errors in the outputs of randomly selected instructions. SASSIFI allows us to select di erent types of instructions to study how error in them will propagate to the application output. As of now (7/22/2016), 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 (Store value)
- Integer add and multiply instructions (IADD-IMAD-OP)
- Single and double precision floating point add and multiply instructions (FADD-FMUL-OP)
- Integer fused multiply and add (MAD) instructions (MAD-OP)
- Single and double precision floating point fused multiply and add (FMA) instructions (FMA-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.

For the I-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. The NVIDIA compiler specifies the maximum number of registers allocated per thread (using *-Xptxas -v* option) and this mode randomly selects a register from that pool for injection. SASSIFI quantifies the masking, DUE, and SDC rates from error injections in allocated registers. These DUE/SDC rates can be further derated by the average fraction of physical registers that are unallocated on a target GPU to obtain AVF of the RF. The average fraction of unallocated registers can be obtained by using performance metrics that measure (1) average fraction of Streaming Multiprocessors (SMs) used in the device and (2) average fraction of warps used per SM. Profiling tools such as *nvprof* can be used to obtain these parameters.

3 What errors can SASSIFI inject?

For the D-mode, SASSIFI can inject the error in a destination register based on the different Bit Flip Models (*BFM*). In the current release (as of 7/23/2016), 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 I-mode, SASSIFI only 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 lock le module is needed to run the injection jobs in parallel either on a multi-gpu system or a cluster of nodes with shared lesystem. Instructions to install the lock le module can be found here [3]. This module is not needed if you want to run injection jobs sequentially on a single-gpu system.
  - (Optional) The nal results can be parsed into an xlsx le using the xlsxwriter module. Instructions to install xlswriter can be found here [4]. The results will be parsed into multiple text les, which can be copied into an excel le to plot and visualize the results.
- SASSI, which can be downloaded from GitHub [5]. SASSIFI is tested using the latest commit (5523d984ad047a272297c1a3 8c63f55c0ad026) on 7/19/2016. 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 pro ling 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/sassi _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 make les (e.g., SASSIFI_HOME/suites/example/simple_add/Make le.

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 les provided in the err_injector directory to avoid keeping multiple copies of the SASSI handler les.

3. **Compile the SASSIFI handlers**: Simply type *make* in $SASSI_SRC/instlibs/src/err_injector. This should create three libraries. The rst one is for pro ling the application and identifying how many injection points exist. The remaining two are for injecting errors during an application run (one each for performing register le and instruction output-level injections).

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_injector" or "make OPTION=rf_injector" to generate binaries for error injection campaigns for the two 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_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 the rf_bfm_list and 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. This 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 decide to use, 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 I-mode injections. If you do not plan to perform I-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 I-mode and across the instructions from the specified instruction group in the D-mode. 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 D-mode. We select the register number within the set of allocated registers for the selected static kernel for the I-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.

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	This applies only to the I-mode injections. The register selected for error injection, but it was never read.
	Written before being read	This applies only to the I-mode injections. The register selected for error injection was overwritten before being read.
	Other reasons	For the I-mode injection, the injected error was consumed but masked later in the application. For the instruction output-level injections, 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 con gurable 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 di erent than the fault-free one.
	Recorded error messages in *stdout*	Error messages are recorded in the *stdout*.
	dmesg error and *stderr* le is di erent	Stderr is di erent, but messages are recorded in the linux kernel (accessed using *dmesg* utility).
	dmesg error and stout le is di erent	*Stdout* is di erent, but messages are recorded in the linux kernel (accessed using *dmesg* utility).
	dmesg error and the output le is di erent	Output le (if it exists for the application) is di erent, but messages are recorded in the linux kernel (accessed using *dmesg* utility).
	dmesg error and application speci c check failed	User-speci ed application speci c (SDC) check failed, but messages are recorded in the linux kernel (accessed using *dmesg* utility).
SDC	Application nishes without crashes, hangs, or failure symptoms but at least one of the outputs of the application is di erent.	
	Stdout is di erent	Text printed in *stdout* is di erent. Output le generated by the application is identical to the fault-free run.
	Output is di erent	The output le generated by the application is di erent than the output generated by the fault-free run.
	Application-speci c check failed	The application-speci c check provided by the user failed.

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 instruction output-level error injections, simply add a *case* in the *switch* statement in the sassi_after_handler function in err_injector/injector.cu, similar to the IADD_IMUL_OP.
- 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 icid_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

I-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.

D-mode: Injection mode in which the register selected for injection is dependent on the instruction executing at the time of injection. This mode allows us to perform targeted error injections on various instruction groups.

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, 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.

[3] lock le 0.12.2 : Python Package Index. https://pypi.python.org/pypi/lockfile.

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

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