SASSIFI: Evaluating Resilience of GPU Applications

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 will grow increasingly important. As soft errors, such as those caused by high-energy particle strikes, form an important fraction of in-field hardware errors, 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 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 condition code and predicate registers, in addition to general-purpose registers and GPU memory. This paper describes our error injection tool and presents some experiments to illustrate some possible lines of analysis. We injected errors into Rodinia benchmark applications and provide results from those experiments showing average detected and silent error probabilities for applications, static kernels, and dynamic kernel invocations. For applications with multiple invocations of the same static kernel, we also show how our tool can be used to study error propagation as a function of the injection time. We also study the effect of errors on condition code and predicate registers.**

***Index Terms*—GPUs, Transient Errors, Silent Data Corruption, Architecture**

I. Introduction

Transient hardware errors from soft errors, such as from 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 or SDCs. When designing a system, it is important to accurately evaluate DUE and SDC rates. Traditionally, hardware-level error injection studies are performed to evaluate how low-level errors (e.g., errors at gate-level) propagate to the architecture level. This approach provides an error derating factor that can be used for full-chip resilience evaluation. However, measuring such low-level error injections is often slow, and simulating the application to completion to understand the effect of the error on program output is difficult. As a result, developing insight into whether and how transient errors can affect application outputs is challenging.

The full application-level transient error derating factor can be obtained through a two-tiered approach. The first tier evaluates the derating factor from the circuit or gate level to the architecture level. The second tier evaluates the derating factor (masking, detection, and SDC rates) for errors at the architecture level that propagate to application output. This second step requires application-level error injection studies with the ability and speed to run many error injection simulations to application completion to evaluate their effect on the program output. Such application-level studies also allow us to understand what application sections are more vulnerable than others, which can help to devise cost-effective error mitigation schemes to improve application resilience. 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 low-level assembly-language instrumentation tool called SASSI overcomes this challenge and provides a platform to perform error injection studies at the architecture level with relatively moderate execution overheads [1]. SASSI provides the ability to instrument instructions in the low-level GPU assembly language (SASS). 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 sensitivity to transient hardware errors by injecting errors into the architecture state of a running GPU. A similar technique used CUDA-GDB [2], [3], but that work lacked the ability to modify predicate and condition code registers and also required complex steps to minimize performance overhead. Some studies investigate GPU vulnerability using ACE analysis [4][5][6] or fault injection [7], but few study the impact of error propagation and masking at the GPU architectural level on application output.

Performing error injections using SASSI requires 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.

Overall, SASSI allows us to model and inject architecture-level errors while GPU workloads are executing and evaluate their effect on application behavior. In this study, we inject single-bit errors in general purpose registers, condition code registers, predicate registers, and store values and observe their effect at the application level for 16 workloads from the Rodinia benchmark suite [8].

Compared to simulation-based approaches, our methodology achieves much greater experimental efficiency. However, we are also limited to injecting errors into architecturally visible state. As a result, we must abstract the effect of lower-level physical faults into our set of injected errors. Our set of injected errors is simplistic, i.e., single-bit flips distributed uniformly over destination values of executing instructions. Injections based on this error set introduce some bias into our results. Nonetheless, we present initial results to illustrate the possible types of analysis with our approach. Specifically, our results show:

- Approximately 79% of errors injected into live GPU architecture state (for our architecture-level error model) have no effect on application execution. Only 7% produce silent data corruptions (SDCs) and the remaining 14% either result in a crash, a hang, or a visible symptom of a failure that can be detected with appropriate error detectors in place. Note that these architecture-level error rates are higher than expected architectural vulnerability factor (AVF) rates because our SASSI-based tool injects into known live architecture state, while AVF measurements also include injections into unused architecture state.
- Even within the same application, kernels vary in SDC susceptibility. For example in *bfs*, we observed nearly $2\times$ differential in SDC rates between two different kernels.
- Different invocations of the same kernel can have different error outcomes. For example, the potential SDC rate increases with increasing kernel invocation index for *pathfinder* and decreases with invocation index for *gaussian* and one of the static kernels in *bfs*.
- Injections into condition code (CC) registers, predicate registers (PR), store values, and general purpose destination registers show different error results. Injections into store values produce significantly higher application crashes, hangs, and failure symptoms compared to injections in CC, PR, and general purpose registers. We did not see any crashes or hangs from PR injections.

II. Background and Related Work

A. Background: SASSI and CUPTI

SASSI is a compiler-based instrumentation tool that runs as the final pass in NVIDIA's production backend compiler and assembler, `ptxas`. Because SASSI is invoked after the original, un-instrumented SASS has already been finalized, injected instrumentation does not disrupt the perceived final instruction schedule or register usage.

SASSI must be instructed *where* to insert instrumentation, and *what* instrumentation to insert. In this paper, we use SASSI to inject instrumentation after all SASS instructions that modify registers or memory. 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 machine (CPU) to both initialize instrumentation counters and to gather their values. CUPTI allows host-side code to receive callbacks when certain important CUDA events occur, such as kernel launches and exits [9]. We use the CUPTI library to initialize counters before kernels launch and to copy profile information off the device after kernels exit.

B. Resilience evaluation studies

Our work uses fault injection at the GPU architectural level to study error propagation at the chip level and the software level. Other chip-level approaches include the use of ACE analysis to derive architectural vulnerability factors [10] and mathematical models to understand the effect of different design and technology parameters [11]. Fault injection can also be performed at the RTL or gate level [12] to understand the effect of errors on combinational and sequential logic and the resulting manifestation of these lower-level faults at the architectural level. Because our work assumes an architectural-level error model, we are not able to capture the exact effect of lower-level faults. However, our approach significantly increases the efficiency of fault injection compared to lower-level fault injection, which not only allows more faults to be injected but also permits the execution of full applications.

Whereas our work uses compiler instrumentation for runtime fault injection, a GPU debugger-based tool [2] can also be used to inject architectural-level errors. Similar to our work, the debugger approach can also inject into the register file. However, our compiler approach also has the ability to inject into condition code and predicate registers. Additionally, we avoid the significant performance degradation of breaking from GPU execution to the host system to inspect and/or modify GPU state.

One prior study looked at measuring Architectural Vulnerability Factor (AVF) of some GPU hardware structures using statistical fault injection to inject faults into register files, local memory, and active mask stack [7]. Our work focuses on application-level resilience evaluations by answering questions such as which static kernel is more vulnerable and which kernel invocation is more vulnerable.

Some work has used accelerated high energy particle beams to measure the failure rates of GPUs [13]. Such studies are able to study the effect of low-level faults with minimal performance overhead but trade off determinism of results, ability to control faults, and visibility into error propagation.

C. Error Model

Because we inject errors at the architectural level, we are studying the vulnerability of the software in allowing lower-level errors to propagate. In some respects, our approach is similar to measuring the Program Vulnerability Factor as defined in [14]. However, our approach uses fault injection instead of ACE analysis. We also only select injection sites from the subset of architectural state that is modified by the application, in contrast to considering all architecturally accessible state. Our results, though not directly translatable into AVF, show how our injected faults affect program outcomes. A full evaluation of AVF requires the distribution of error patterns at the architectural level based upon the propagation of errors through the microarchitecture.

III. SASSIFI Framework

SASSIFI is a SASSI-based error injection approach that injects transient errors in ISA visible state such as general purpose registers, store values, predicate registers, and condition code registers. SASSIFI 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.

Error Model: With the aim of modeling soft-errors in latches and unprotected SRAM structures (e.g., pipeline buffers) at architecture level, we inject bit-flips in outputs of executing instructions. Specifically, we inject a single-bit flip in one of the destination registers of an executing instruction per application run. We exclude instructions that are predicated out. If the destination register is a general purpose register (32-bit value) or a condition code (4-bit value), one bit is randomly selected to be flipped. For predicate registers, we only flip one of the destination predicate bits that are written by the instruction. For store instructions, we flip a randomly selected bit in the stored value.

Since we inject errors directly into live state (destination registers), our error model does not account for various masking factors in the lower layers of the hardware stack such as circuit-, gate-, and microarchitecture-level masking as well as masking due to errors in architecturally untouched values. Our model does not directly provide insight into AVF of the applications – our results in Section IV show the breakdown in program outcomes (but does not imply any specific AVF, or enable a comparison across different programs). A full evaluation of AVF would need to (1) couple into a lower level microarchitecture error model, and (2) account for pure random injection of fault state.

Low-level transient errors may or may not propagate to the architecture level as single-bit flips in destination values. Also, our model uniformly selects dynamic instructions for error injections. Since some instructions may be more amenable to soft-errors than others (e.g., based on the number of latches/flip-flops they exercise), we need further research to derive accurate architecture-level transient error models to better represent low-level transient errors and to bias injection instruction selection.

Profiling and identifying the error injection space: For the profiling step, we use SASSI to instrument after all instructions, except for the instructions that are predicated out. We collect and pass the register and memory information for each instrumented instruction to the handler, which records the state modifications so that an off-line tool can stochastically select the error injection sites. Specifically, we collect the following information to identify the error injection space: (1) static kernel names in an application, (2) number of times each kernel executes, (3) number of threads per kernel invocation, and (4) number of dynamic instructions per thread that are not predicated out and either write to a register or a memory location. We use CUPTI to collect (1) and (2) and instrument the instructions using SASSI to collect (3) and (4).

TABLE I
Error injection outcomes.

Category	Explanation
Masked	Application output is same as the error free output. No error symptom is observed.
Crashes	Application exits with non-zero exit status.
Hangs	Executions that do not terminate within an allocated threshold, which is 3× the fault-free runtime in our study.
Failure Symptoms	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.
Stdout only different	Application finishes without crashes, hangs, or failure symptoms but the output of *stdout* is different. Output file generated by the application is identical to the fault-free run.
Output different	Application finishes without crashes, hangs, or failure symptoms but the output file generated by the application is different than the output generated by the fault-free run. Most of our applications produce a single output file.

Statistically selecting error injection sites: Using this information, we randomly select 1,000 dynamic instructions for error injections among all dynamic instructions across all kernel invocations per application. At 95% confidence intervals, the measured potential SDC rates have error bars under 3%. For each selected dynamic instruction, we randomly select a destination value (e.g., general purpose register or predicate register) and then select the bit to be flipped, which is also randomly selected from the bits that are being written by the instruction (e.g., one bit is selected among the predicate register bits that are being written by the instruction or one bit among the 32 bits of a general purpose register). Specifically, an error injection site is a tuple consisting of the static kernel name, dynamic kernel invocation ID, thread ID, dynamic instruction count, seed to select a destination value, and seed to select the bit for injection. This step is performed on the host CPU.

Error injections runs: In the last and the most important step, we instrument the same set of instructions as in the profiling step and use the instrumentation handler to inject the error into the location selected by the stochastic process. We inject one error per application run and monitor for crashes, hangs, and output corruption. In each injection run, we check if the dynamic invocation count of the kernel has been reached using CUPTI. If so, we copy the remaining error site tuple into the device memory. During kernel execution, we check if the current thread is the selected thread for injection in the instrumentation handler. For the selected thread, we maintain a counter and check if the dynamic instruction that just executed is the selected instruction for injection. For this instruction, we inject the error based on the instruction and destination register being selected. We select the bit using the seed that is passed as input.

After the error injection, the application is then executed 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, differences in *stdout/stderr*, and program output (typically stored in a file) from that of the error-free runs. Table I explains how we categorize error injection outcomes.

TABLE II
RODINIA APPLICATIONS.

Benchmark	Domain	# Static Kernels	# Dynamic Kernel Calls
b+tree	Search	2	2
backprop	Pattern recognition	2	2
bfs	Graph algorithm	2	24
pathfinder	Grid traversal	1	5
kmeans	Data mining	2	3
nn	Data mining	1	1
streamcluster	Data mining	1	1611
lud	Linear algebra	3	46
gaussian	Linear algebra	2	2046
mummergpu	Bioinformatics	2	2
nw	Bioinformatics	2	255
lavaMD	Molecular dynamics	1	1
hotspot	Physics simulation	1	1
heartwall	Medical imaging	1	10
srad_v1	Image processing	6	502
srad_v2	Image processing	2	4

Crashes, hangs, and failure symptoms can be categorized as detected unrecoverable errors (DUEs). Injections marked as *stdout only different* and *output different* can be categorized as potential silent data data corruptions (SDCs).

IV. EVALUATION

Our experimental flow targets NVIDIA Tesla K20 and Telsa K10 G2 GPUs with 5GB and 8GB memory, respectively. Our setup uses display driver version 340.29 and the CUDA 6.5 toolkit. The host system is an Intel Core 2 Quad CPU Q6600 with 4GB of memory. Table II lists the Rodinia applications used in this study [8].

As mentioned above, we performed 1,000 error injection runs per application using our error model that injects single-bit transient errors directly into the destination values of executing instructions. Figure 1 shows how different applications behave with architecture-level errors. The figure shows that approximately 79% of injected errors on average did not have any effect on the program output. Only 10% resulted in crashes or hangs. Approximately 4% of the injections showed symptoms of failures that can be categorized as potential crashes with appropriate error monitors. These categories are unsuccessful kernel executions (detected by comparing kernel exit status with *cudaSuccess*) or explicit error messages in stdout/stderr (e.g., Error: misaligned address). The remaining injections corrupt some application output (stdout, stderr, or program defined output file). We categorize such cases as potential silent data corruptions (SDCs). We observed that 1.5% of the injections showed differences only in stdout/stderr when compared to error-free executions without corrupting the program output file, which may be acceptable for some applications. Lastly, only 5.4% showed differences in the program output file.

A. Comparing error sensitivity of different static kernels

Several of our studied applications have more than one static kernel that executes for a significant fraction of application runtime. Data obtained through our error injection campaign allows us to study the sensitivity of different static kernels to architecture-level single-bit flips. Figure 2 shows our findings. The height of the bars shows the number of injection experiments performed per static kernel per application (these numbers sum to 1,000 per application). Since we select instructions

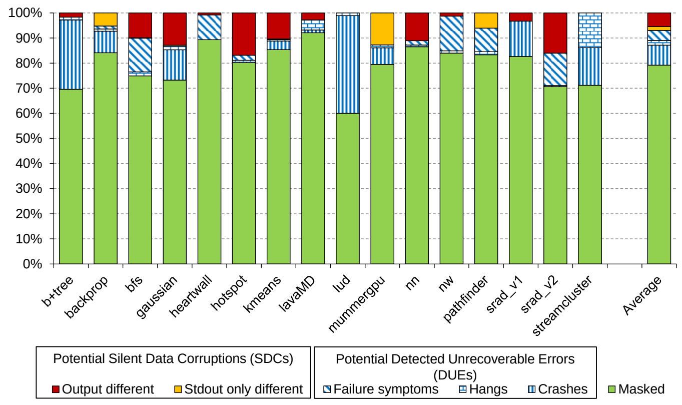

Fig. 1. Error injection outcomes of different applications. Each bar shows the results obtained from 1,000 error injection runs using our error model that injects single-bit transient errors directly into the destination values of executing instructions.

for error injection randomly across all the application dynamic instructions, the heights of the bars can also be approximated as the relative number of dynamic instructions per static kernel across all its instantiations.

For some applications, not all static kernels in an application appear in this figure. The number of dynamic instructions executed in those kernels is small enough compared to other kernels in that application such that no error injection site was selected from that kernel (we randomly select 1,000 dynamic instructions among all dynamic instructions across all dynamic invocations of kernels).

Figure 2 shows that different static kernels within an application behave differently for several applications. For example, the bfs application has two kernels named Kernel and Kernel2, with the second kernel (Kernel2) experiencing more potential SDCs (output differences) and fewer failure symptoms. Similarly, for srad_v2, we observe that the srad_cuda_2 kernel experiences more output differences (potential SDCs) than the srad_cuda_1 kernel. For these two applications, the kernel that experiences the higher number of potential SDCs executes for a relatively smaller amount of time on the GPU, which suggests that application-level error mitigation schemes such as partial or full thread-level redundancy can be cost effective.

We also observe that for mummergpu, the kernel mummergpuKernel experiences significantly more potential SDCs (*stdout only different*) compared to the other kernel in that application. Lastly, for srad_v1, srad is the only static kernel that experienced significant output differences.

The number of injections per static kernel (heights of the bar) for most applications in Figure 2 are at least 300, which is significant to study the trends. However, further experimentation is required to develop the insights behind these observations.

B. Comparing error sensitivity of different dynamic kernel invocations of same static kernel

We also study how error outcomes change with different dynamic kernel invocations of the same static kernel. Since some applications have many dynamic kernel invocations, we do not have a sufficient number of injections per dynamic kernel invocation for all applications. However, for several applications, we do have 10s to 100s of injections per dynamic

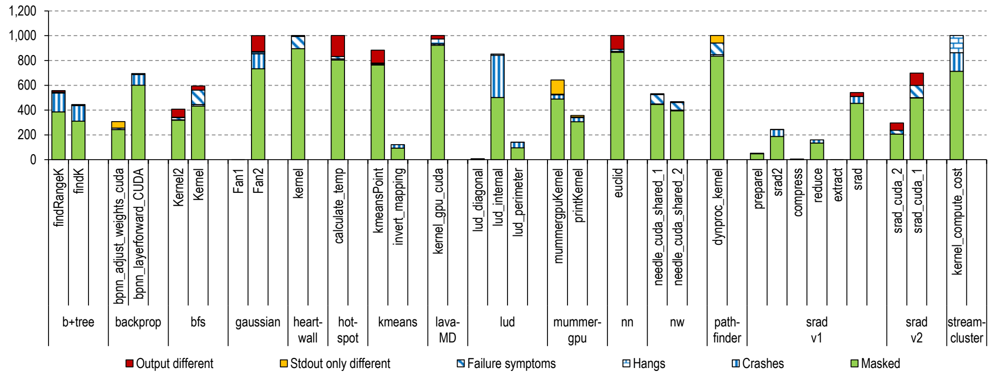

Fig. 2. Breakdown of the error injection outcomes per static kernel of different applications is shown here. Height of each bar (y-axis) shows the number of injections performed per static kernel.

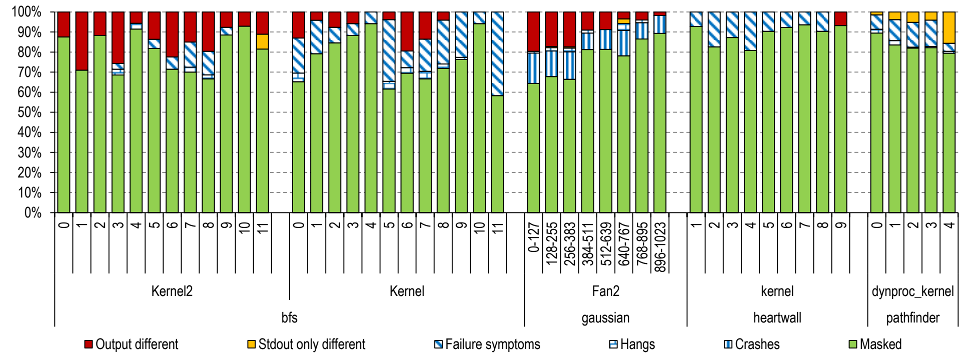

Fig. 3. Error outcomes change with the kernel invocation index for different static kernels in bfs, gaussian, heartwall, and pathfinder.

kernel invocation to observe trends. We show these results in Figure 3 for bfs, gaussian, heartwall, and pathfinder with the x-axis as the dynamic invocation index for each static kernel, which is equal to the number of dynamic kernel invocations of that static kernel until the point of error injection.

For bfs, one static kernel experiences more output differences than the other. Figure 3 shows how error outcomes differ between different dynamic kernel invocations for the two static kernels. The two kernels (*Kernel* and *Kernel2*) are called back to back in a loop that executes 12 times for the input we selected. Although no pattern emerges for *Kernel2*, for *Kernel*, the number of output differences decreases with the dynamic invocation index, except for the invocation index 6 where we observe a steep increase.

We also observe a similar trend for gaussian (see Figure 3). However, the number of potential SDCs (output differences) decreases significantly with the dynamic invocation index, and the number of masked cases increases. We hypothesize that errors in the early invocation indices are exaggerated by subsequent computation, causing this effect.

We also observe that for some application kernels, e.g., *kernel* in heartwall, there are hardly any noticeable differences in outcomes across invocations (Figure 3). On the other hand, for pathfinder we observed that the number of potential SDCs (cases with stdout only different) increases with the dynamic kernel invocation index. This effect may be due to the application-level masking effects for errors in early invocation indices, e.g., if an error is injected early in the execution, it has more opportunity to get masked.

C. Sensitivity to error injections in different destination values

Here we show the error outcomes from single-bit error injections in condition codes registers (CC), predicate registers (PR), store values, and others general-purpose destination registers. Recall that we inject errors into one of the randomly selected destination values of an executing instruction. Since the number of instructions that write to a CC, PR, or a store value to memory is relatively small compared to the number of instructions that write to general purpose registers, the number of injections into CC, PR, and store values are significantly lower. Nonetheless, the number of injections in CC, PR, and store instructions across all the applications are 612, 1424, and 569, respectively, which we believe is sufficient to observe trends.

We also observed that errors in predicate registers do not result in crashes, hangs, or other failure symptoms. This effect is likely due to the way predicate registers are used, which is primarily to nullify a single instruction or multiple instructions in a thread. Errors in these bits can either convert instructions to null operations or vice versa.

Lastly, we observe that errors in store values produce more potential SDCs when compared to errors in general-purpose registers, PR, or CC bits. They also experience much higher crashes or failure symptoms (or DUEs). Our hypothesis is that applications use store values for control and address computations as well (not just storing program values that are used in pure data flow), thus increasing DUEs.

D. SASSIFI Slowdowns

We observed modest application level slowdowns compared to uninstrumented runs for our error injection runs. We only show slowdowns for injection runs because these are run 100s

TABLE III
SLOWDOWNS FOR RODINIA APPLICATIONS.

Benchmark	Slowdowns (×) app	Slowdowns (×) kernel	Benchmark	Slowdowns (×) app	Slowdowns (×) kernel
b+tree	1.7	–	kmeans	1.9	105
backprop	4.7	240	lavaMD	13.2	451
bfs	1.6	141	lud	3.7	67
gaussian	5.0	40	mummergpu	1.1	101
heartwall	137.5	291	nn	4.1	–
hotspot	2.0	789	nw	4.7	54
pathfinder	1.7	660	streamcluster	23.9	347
srad_v1	20.3	–	srad_v2	1.7	552

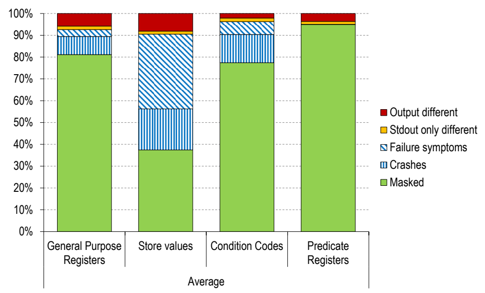

Fig. 4. Outcomes of injections into CC, PR, store, and general purpose destination registers averaged across the studied applications.

to 1000s of times per application, as opposed to profiling runs that are run only once. Table III shows the application- and kernel-level slowdowns for the injection runs that result in masking (excluding early terminations due to crashes and extra time needed for hangs). We observed 1.1× to 137.5× slowdowns at the application level; at the kernel level the slowdowns were much higher, ranging from 40× to 789×.

Since SASSI instrumented all instructions in our setup, the kernel-level slowdowns are significant (due to added instructions and cache pollution). This overhead can be significantly lowered by instrumenting only the static instructions that are selected for error injections, which will most likely be a small fraction of all static instructions.

V. CONCLUSIONS AND FUTURE WORK

This paper describes the SASSIFI tool and presents initial fault injection experiments with SASSIFI that illustrate the utility of the framework for analyzing error propagation within a GPU while executing real applications. We present a preliminary set of results as an example of the types of results and analysis possible with SASSIFI. Based on this work, we plan to improve the quality of results and to study other aspects of error propagation, especially at the software level.

One very interesting line of analysis involves extracting application characteristics that correlate with either detected or silent errors. For example, certain programming constructs or design patterns may be associated with greater error propagation. Identification of those constructs or patterns allows programmers to adjust their code to account for acceptable levels of errors and overhead. This information may also allow selection of highly vulnerable code sections to be protected by software-implemented error detection and correction methods.

Our results show that different invocations of the same kernel experience different error masking. While our current experiments are not sufficient to explain this effect, we expect that deeper experimentation illuminate the causes. For example, by inserting code to check for error propagation at the end of each dynamic kernel, we can more precisely study the effect of inter-kernel error propagation. This type of study might permit the identification of a subset of dynamic kernel invocations that provide sufficient error propagation with acceptable overheads.

The experiments described in this paper assume a simplistic error model, where lower-level faults propagate to the architectural level as single-bit flips in destination registers and where all instructions are equally likely to be injected with a soft error. Deviation of this error model from reality introduces inaccuracies into the SASSIFI results. Therefore, analysis of the pre-architectural error propagation is needed to derive a more realistic set of error patterns for SASSIFI injections. This analysis may involve lower-level fault injection to derive more precise error patterns or less resource-intensive modeling for more approximate error models.

REFERENCES

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

[2] 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, pp. 221–230.

[3] NVIDIA. (2014, August) CUDA-GDB :: CUDA Toolkit Documentation. [Online]. Available: http://docs.nvidia.com/cuda/cuda-gdb/index.html

[4] J. Tan, N. Goswami, T. Li, and X. Fu, "Analyzing soft-error vulnerability on gpgpu microarchitecture." in *Proceedings of the International Symposium on Workload Characterization (IISWC)*, 2011, pp. 226–235.

[5] 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*, March 2013.

[6] M. Wilkening, V. Sridharan, S. Li, F. Previlon, S. Gurumurthi, and D. R. Kaeli, "Calculating architectural vulnerability factors for spatial multi-bit transient faults," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*. IEEE, 2014, pp. 293–305.

[7] 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*, April 2012.

[8] 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 International Symposium on Workload Characterization (IISWC)*, October 2009, pp. 44–54.

[9] NVIDIA. (2014, November) CUPTI :: CUDA Toolkit Documentation. [Online]. Available: http://docs.nvidia.com/cuda/cupti/index.html

[10] 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, pp. 29–42.

[11] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi, "Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic," in *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, June 2002, pp. 389–398.

[12] 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, November 2005.

[13] D. A. G. Oliveira, P. Rech, H. M. Quinn, T. D. Fairbanks, L. Monroe, S. E. Michalak, C. Anderson-Cook, P. O. A. Navaux, and L. Carro, "Modern GPUs Radiation Sensitivity Evaluation and Mitigation Through Duplication With Comparison," *IEEE Transactions on Nuclear Science*, vol. 61, pp. 3115–3122, December 2014.

[14] V. Sridharan and D. Kaeli, "Eliminating Microarchitectural Dependency from Architectural Vulnerability," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, February 2009, pp. 117–128.