PCFI: Program Counter Guided Fault Injection for Accelerating GPU Reliability Assessment

Fritz G. Previlon, Charu Kalra, Devesh Tiwari and David R. Kaeli
Northeastern University, Boston, MA, USA

***Abstract*—Reliability has become a first-class design objective for GPU devices due to increasing soft-error rate. To assess the reliability of GPU programs, researchers rely on software fault-injection methods. Unfortunately, software fault-injection process is prohibitively expensive, requiring multiple days to complete a statistically sound fault-injection campaign.**

Therefore, to address this challenge, this paper proposes a novel fault-injection method, PCFI, that reduces the number of fault injections by exploiting the predictability in fault-injection outcome based on the program counter of the soft-error affected instruction. Evaluation on a variety of GPU programs covering a wide range of application domains shows that PCFI reduces the time to complete fault-injection campaigns by 22% on average, without sacrificing accuracy.

***Index Terms*—reliability, fault injection, gpu, soft errors, transient faults**

I. Introduction

GPUs have disrupted multiple domains including HPC, machine learning, autonomous navigation, and scientific computing due to their unprecedented performance. However, due to an increased number of errors in GPU execution, their reliability is starting to outweigh their performance benefits [1]. Among all faults types and causes, radiation-induced transient faults produce the highest rate of failures in today's computing systems [2]. Large-scale HPC systems experience transient faults at an order of minutes [3], [4]. Therefore, reliability has become a first-class design objective for GPU devices.

The first step toward improving the reliability of GPUs is quantifying and assessing the vulnerability of GPU programs toward transient faults since these faults affect different programs differently [1]. To assess the reliability of GPU programs, researchers rely on software based fault injection campaigns. A fault injection campaign consists of running a program a large number of times where during each run, a fault is injected at a randomly chosen point.

Unfortunately, the software fault-injection process is prohibitively expensive since the number of faulty runs in a fault injection campaign needs to be sufficiently high to produce statistically significant results. Previous research papers have reported the number of fault injection runs to be more than 60,000 for achieving high accuracy [5], [6]. Some researchers have limited the number of fault injection runs to be of order of 10,000 to reduce the overall campaign time and obtain the error margins close to 1%, for a 95% confidence interval [7] [8]. Assuming a single run of an application takes as small as 15 seconds on a GPU, it will take more than 45 days on a single GPU to perform 10,000 fault injection runs for the 26 benchmarks considered in this study. As the execution time of a GPU program increases, as is the case for long-running GPU-based HPC applications. This overhead makes reliability assessment impractical or even infeasible in some cases. *To address this challenge, this paper proposes a novel fault-injection method, PCFI, that reduces the number of fault injection runs needed during a fault injection campaign to assess the vulnerability of GPU programs to soft errors, without comprising the accuracy of results.*

PCFI is a conveniently chosen acronym for program counter (PC) guided fault injection. PCFI is pronounced as *Peace-ify*.

II. Background

When a transient fault becomes visible at the software level, it may produce three major outcomes: *masked, detected and unrecoverable error (DUE),* and *silent data corruption (SDC).* Masked outcome means that the fault has no adversarial impact on the output of the program. When the fault may be detected through assertions, by the OS, or other hardware supported checks, but can not be corrected, then it is commonly referred to as a detected and unrecoverable error (DUE). Finally, when a fault causes a program to produce an erroneous output without alerting the user of the incorrect result, it is referred as a silent data corruption (SDC).

This work performs reliability assessment of GPU programs by capturing the effects of injected faults on the program execution (i.e., masked, SDC, and DUE). We evaluate the resilience of key elements in the instruction execution datapath (e.g., arithmetic logic units, load-store units) of a GPU in the presence of single-bit transient faults. We inject faults in the destination registers of instructions that write to a general purpose register in the GPU. Our fault model is a standard model – used by other recent GPU reliability assessment works [1], [8], [9].

III. PCFI: Design and Implementation

Widely-used GPU fault injectors take the following 3 steps to perform fault injection: 1) They identify all possible vulnerable locations, i.e., all the dynamic instructions of the program under evaluation; 2) They randomly and uniformly select a long list of faults (up to 60,000), consisting of threads and instructions within a thread for fault injection, to cover different program phase behavior in space and time; 3) They run the program with each fault in the list. We use the fault-list obtained from this method as the baseline fault list that PCFI prunes.

PCFI exploits the following two observations to reduce the number of fault-injected runs from the base list. First, GPU programs tend to have a small set of static instructions (PCs) that often constitute a significant fraction of total dynamic instructions. Consequently, these PCs also account for a major fraction of injected faults in the baseline case. Second, the

978-3-9819263-2-3/DATE19/©2019 EDAA

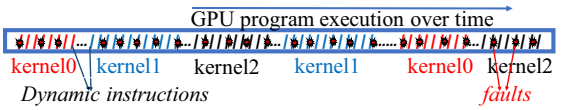


Fig. 1: Traditional fault injection methods uniformly and randomly distribute faults across a program's dynamic instructions.

outcome of injected faults in many frequently executed PCs remain identical across different dynamic instances of the same static instruction.

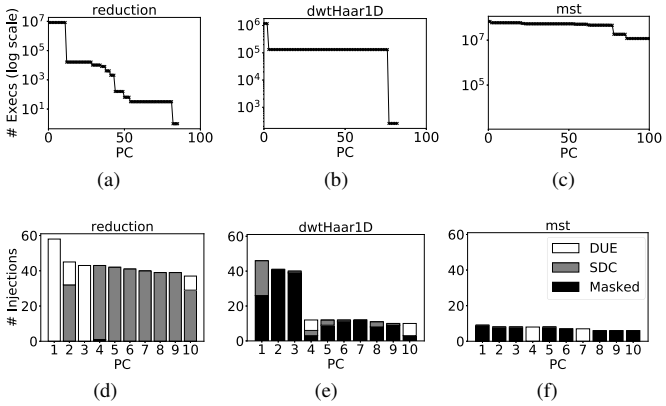
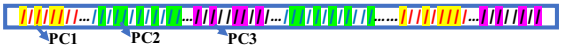


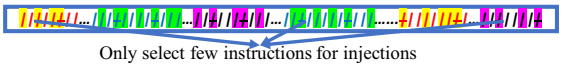
Fig. 2: Applications from different domains, *reduction (red)* (data processing), *dwtHaar1D* (signal processing), and *mst* (graph traversal), show that (1) few static instructions (PCs) dominate overall dynamic execution of instructions, (2) vulnerability outcomes in highly frequently PCs are likely to remain across injections for many PCs.

Next, we provide quantitative evidences to support our observations. Figs. 2a, 2b, and 2c show the execution breakdown across each PC in three programs. These results show that some PCs are highly dominant. That is, their execution frequencies can be an order of magnitude greater than other PCs in the same application. Using dynamic profiling of the *red*, *dwtHaar1D*, and *mst* applications, we found that 86, 83, and 615 unique PCs are executed, respectively. However, very few PCs dominate most of the execution in these programs. For example, in *red*, 99% of the dynamic instructions were different occurrences of only 12 PCs.

Figs. 2d, 2e, and 2f show the number of faults into unique PCs after a full fault injection campaign with SASSIFI, for a 500 randomly and uniformly distributed fault baseline. Many of the top 10 PCs obtain a significantly greater number of faults. For example, in *red*, top 10 PCs account for 80% of total faults out of 500. Second, we observe that the outcome of injected faults is identical for many of the frequently executed PCs for all three benchmarks. To exploit this opportunity, one could limit the number of injections in these "friendly"-PCs and extrapolate the results for each PC according to its respective execution frequency. We note that not all PCs demonstrate this behavior due to varying control flow behavior during the lifetime of the program. For such PCs, the faults yield different outcomes depending on the control flow of the application and hence, they should be given their full share to fully capture their behavior.



(a) In GPU programs majority of dynamic instructions is dominated by a handful of static instructions (PCs). In this example, 3 PC's dominate the dynamic execution of the program.



(b) PCFI limits the number of injections into instances of individual static instructions.

Fig. 3: PCFI approach for PC-guided fault injections.

Leveraging these insights, PCFI identifies frequently executed PCs via a profile run and carefully limits the number of fault injections in the those PCs if the outcome of a fault in those PCs does not change across different dynamic execution instances. Fig. 3b shows that PCFI limits the number of injections in PCs on the baseline shown in Fig. 3a, where most of the dynamic instructions in this example are different dynamic occurrences of 3 dominant PCs. Our evaluation results show that limiting the number of injections based on history outcomes does not compromise the accuracy of results.

PCFI Implementation Details

Next, we describe how PCFI is implemented in an open-source GPU fault injector, SASSFI [1]. The steps describe below can be adopted for any generic GPU fault injector and are largely independent of how GPU fault injector is implemented.

Profiling Stage: First, PCFI performs a profiling step which records the count of all dynamic instances of each static instruction in the given GPU program. At the end of this step, PCFI identifies all the static instructions (PCs) that are executed most frequently and their respective execution frequency.

Fault List Generation Stage: In this step, PCFI generates the fault list. PCFI generates these faults such that they are randomly and uniformly distributed across the dynamic executions of each PC. This also ensures that PC behavior across all different program phases is captured. The total number of faults is provided by the user. The user decides the number based on the error margin and confidence level she wants to achieve.

We note, that up to this stage, PCFI would yield the same vulnerability profile (i.e., the same ratio for each type of outcome of fault injections) as the traditional methodology since the fault list has not been pruned yet.

Fault Injection Stage: In this step, PCFI instruments the fault injection handler in SASSIFI to inject a fault based on the PC value. Once the correct execution count for a specific PC is reached, PCFI handler injects a fault in a destination register of the instruction.

To reduce the number of fault-injected runs, PCFI keeps track of the vulnerability outcomes of faults injected in each PC. If a particular PC is a frequently executed PC (i.e., it receives greater number of faults than a threshold), then PCFI first randomly and uniformly picks "threshold" number of faults corresponding to this PC and executes these runs. If the outcomes of these runs are identical (e.g., all producing DUE), then PCFI eliminates further fault injections to this particular

PC and hence, reduces the number of fault-injected runs. If the outcomes differ for first "threshold" number of faults, then, PCFI continues to inject faults. PCFI chooses the threshold to be 1% of total number of faults in the base case.

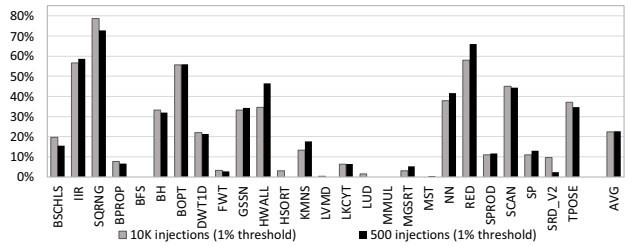


Fig. 4: Percentage of reduction of the total number of injections with PCFI.

IV. Evaluation and results

We evaluate our methodology on a mix of 26 applications, selected from 2 well-known benchmark suites for GPU applications, the Rodinia suite of applications [10] and the CUDA-SDK [11]. These applications are representative across a range of application domains, and been widely used for GPU performance and reliability assessments [9].

We present our results in two categories. First, we look at the fault list reduction and corresponding time savings achieved from the baseline fault list (i.e., 10K random fault injections). Then, we will evaluate the accuracy of our methodology by comparing our results to a traditional fault injection campaign performed using SASSIFI.

A. Fault list reduction

We compare our results to two different baselines: 10,000 random injections (a 95% confidence level, with a 0.98% error margin) and 500 random injections (a 95% confidence level, with a 4.38% error margin).

Fig 4 shows the percent reduction in the number of faults injected. Our results show that PCFI is effective in both cases as compared to the baseline case of 10K injections, and even when we reduce the number to 500 injections. PCFI achieves an average fault list reduction of 22.38% when starting with 10K injections, resulting in an average of 7,762 faults per application. Starting with 500 injections, PCFI achieves 22.68% reduction and 387 faults per application.

The magnitude of the fault list reduction changes based on the specific workload characteristics. For our 10,000 fault baseline, the fault list reduction varies between 0 and 78.8% (*SQRNG*).

Our analysis reveals that the wide range in fault list reductions can be attributed to the following factors:

1) **The Number of Dominant PCs in the Application**: If there are fewer dominant PCs in the application, it means that the majority of the execution is confined to a small set of the PCs. This increases the opportunity for PCFI to reduce the number of injections that target those PCs.

2) **Execution Frequency of the Dominant PCs**: Fewer dominant PCs executing significantly more often than other PCs offer a bigger margin for fault reduction through PCFI. In a case where the execution is spread out across many dominant PCs, PCFI will not offer much savings. It is important to note that a GPU programming best practice is to develop code that has good temporal locality, where most of the run time is spent in a limited amount of code [12].

The application *reduction (RED)* experiences a very significant fault list reduction (58% for 10K faults, and 66% for 500 faults). We find that this application spends 99% of its dynamic execution in 12 unique PCs (out of a possible 86 unique PCs present in the application), with the number of executions 3 orders of magnitude greater than the remaining PCs in the application (see Fig. 2a). This application is a classic parallel GPU application, that can greatly benefit from using PCFI.

On the other hand, the execution time for the *minimum-spanning-tree (MST)* application is spread out fairly evenly across a large number of unique PCs (see Fig. 2c). In this program, 95% of dynamic instructions are confined to over 233 unique PCs (out of a total of 615 unique PCs). This application does not present many opportunities to use PCFI for fault list reduction.

TABLE I: Time savings using PCFI

N	Traditional	PCFI	N	Traditional	PCFI
10K	48.15 days	*36.16 days*	500	57.73 hrs	*42.57 hrs*

Time savings: Table I shows the time it takes to perform a fault injection campaign for the 26 applications in our testing framework. We compare the time expended using the traditional approach vs. PCFI. Even with an already small number of injections per application (i.e., 500), PCFI offers significant time savings of 15 hours. This time savings can rapidly increase when researchers perform multiple fault injection runs, as mitigation efforts need to be evaluated and new resilience mitigation strategies are being devised.

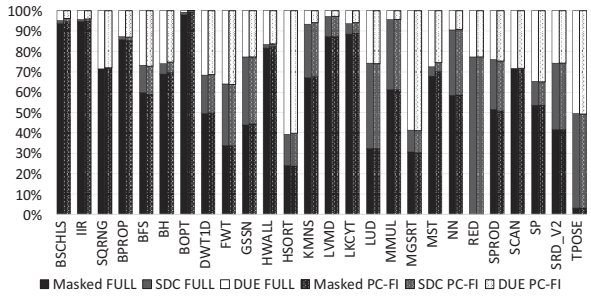
B. PCFI accuracy

Next, we evaluate the accuracy of the fault injection campaign performed using PCFI. In Figure 5, we show the results of a fault injection campaign performed using PCFI, along with a traditional fault injection campaign. These results show that PCFI achieves the same results as the traditional fault injection campaign. They also show that for all 26 applications, a fault injection campaign performed with PCFI can be consistently well within the error margin for all fault outcome categories. We also show the average and maximum error between PCFI results, and the results from a traditional fault injection in Table II. Notably, PCFI achieves average accuracy error of less than 0.55 across all cases, and maximum accuracy error of 2.49%. The minimum error is 0% in all cases.

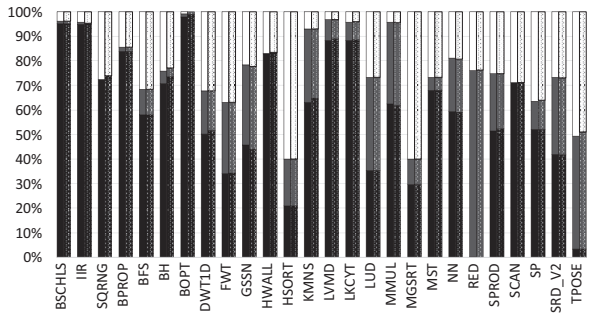
To demonstrate that PCFI produces better results than a traditional fault injection strategy using a reduced fault list (the same number of faults as PCFI), but randomly injected, we selected the 3 applications which achieve the smallest number of faults, starting with the original fault list of 500 injections:

TABLE II: Max and Average error with PCFI.

Outcome Type	N=10K		N=500	
	Max Error	Avg Error	Max Error	Avg Error
Masked	2.49%	0.46%	2.66%	0.53%
SDC	0.90%	0.17%	1.82%	0.45%
DUE	2.04%	0.42%	1.82%	0.32%



(a) Starting with 10K faults



(b) Starting with 500 faults

Fig. 5: Results for PCFI compared to a traditional fault injection campaign.

TABLE III: Comparing two techniques: 1) PCFI and 2) traditional fault injection with a reduced fault list (as offered by PCFI with 500 faults) against a baseline of 500 fault injection campaign.

		PCFI			Traditional		
App	Outcome	Avg. Error	Max Error	Std Dev	Avg Error	Max Error	Std Dev
SQRNG	Masked	1.145%	1.731%	0.406%	3.124%	4.591%	1.135%
	SDC	0.000%	0.000%	0.000%	0.000%	0.000%	0.000%
	DUE	1.145%	1.731%	0.406%	3.124%	4.591%	1.135%
RED	Masked	0.163%	0.285%	0.109%	0.527%	1.485%	0.536%
	SDC	1.196%	1.531%	0.237%	2.363%	5.245%	2.450%
	DUE	1.148%	1.449%	0.272%	2.889%	5.553%	2.287%
IIR	Masked	0.887%	1.452%	0.373%	2.118%	2.936%	1.022%
	SDC	0.476%	1.000%	0.385%	0.780%	1.899%	0.713%
	DUE	0.604%	1.246%	0.548%	1.560%	2.828%	0.883%

SobolQRNG (136 faults), *reduction* (170 faults) and *IIR* (207 faults). We evaluate whether the reduced number of injections obtained with PCFI could provide similar error margins as compared with a traditional methodology. *PCFI uses 136, 170, and 207 faults, whereas the baseline uses a fault list of 500. Now, these faults (136, 170, and 207 faults) are being injected randomly in the respective benchmarks to demonstrate that PCFI is more effective than randomly pruning the fault list by the same amount.*

We perform a fault injection campaign for each approach (PCFI and the traditional method with a reduced fault list) 5 times to check the consistency of the results. For each campaign, we take the difference between the results of the technique and the results of a baseline using 500 random fault injections. We find the differences in the maximum, average, and standard deviation across the 5 fault injection campaigns. The baseline is obtained by taking the average results of 5 fault injection campaigns of 500 faults. The results are presented in Table. III, showing PCFI consistently produced similar results as the baseline (max error being 1.73%), while the equivalent number of faults in the traditional methodology shows a much higher variation over the five fault injection run (max error being 5.55%). These results indicate that, compared to the full traditional methodology, PCFI offers a lower margin for error (2x better in many cases), and a more consistent assessment (lower standard deviation).

V. Related Work

In this section, we review the most relevant prior studies related to our research. Nie *et al.* have proposed a multi-stage technique to prune the GPU fault injection sites [5]. Their work assumes that all threads in a block and common instruction blocks are likely to have similar resilience characteristics. Through our evaluation on a diverse set of regular and irregular GPU applications, we show that this assumption may not always hold true as faults in the same PC can often lead to different outcomes.

Kaliorakis *et al.* perform a fault list reduction by running an ACE-like analysis and removing the faults that fall into the non-vulnerable intervals of a program [6]. Our work differs from theirs in that we focus on fault injection on live architectural state and already accounts for any non-vulnerable portions of a program.

We note that these works are simulation-based, and hence, may not capture long and representative program phase behavior.

VI. Conclusion

This work introduced a new fault-injection method, PCFI, that significantly reduces the number of fault injection runs needed during a fault injection campaign without comprising the accuracy of vulnerability assessment of GPU programs. This is the first work that exploits PC based behavior for predicting the vulnerability of instructions toward soft-errors and thereby, reducing the time spent in fault injection campaigns by up to 78% for a wide variety of GPU benchmarks. The proposed method is implemented in SASSIFI, an open-source GPU fault injector. This methodology can help researchers accelerate real-system fault-injection studies and evaluation of resilience mitigation strategies.

References

[1] S. Hari *et al.*, "Sassifi: An architecture-level fault injection tool for gpu application resilience evaluation," in *ISPASS 2017*.
[2] R. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," *IEEE Transactions on Device and Materials Reliability*.
[3] L. B. Gomez *et al.*, "Gpgpus: How to combine high computational power with high reliability," in *DATE 2014*.
[4] D. Tiwari *et al.*, "Understanding gpu errors on large-scale hpc systems and the implications for system design and operation," in *HPCA 2015*.
[5] B. Nie *et al.*, "Fault site pruning for practical reliability analysis of gpgpu applications," in *MICRO 2018*.
[6] M. Kaliorakis *et al.*, "Merlin: Exploiting dynamic instruction behavior for fast and accurate microarchitecture level reliability assessment," in *ISCA 2017*.
[7] R. Leveugle *et al.*, "Statistical fault injection: Quantified error and confidence," in *DATE 2009*, April 2009, pp. 502–506.
[8] G. Li *et al.*, "Understanding error propagation in gpgpu applications," in *SC 2016*, Nov 2016.
[9] C. Kalra *et al.*, "Prism: Predicting resilience of gpu applications using statistical methods," in *SC 2018*.
[10] S. Che *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," in *IISWC 2009*.
[11] NVIDIA, "NVIDIA, CUDA SDK, V7.0."
[12] ——, "CUDA C best practices guide," 2018. [Online]. Available: https://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf