Combining cluster sampling and ACE analysis to improve fault-injection based reliability evaluation of GPU-based systems

Combining Cluster Sampling and ACE analysis to improve fault-injection based reliability evaluation of GPU-based systems

Alessandro Vallero and Stefano Di Carlo
Politecnico di Torino, Control and Computer Engineering Department
Torino, Italy
Email: {name.surname}@polito.it

***Abstract*—Computing capability demand has grown massively in recent years. Modern GPU chips are designed to deliver extreme performance for graphics and for data-parallel general purpose computing workloads (GPGPU computing) as well. Many GPGPU applications require high reliability, thus reliability evaluation has become a crucial step during their design. State-of-the-art techniques to assess the reliability of a system are fault injection and ACE analysis. The former can produce accurate results despite eternal time while the latter is very fast but it lacks accuracy of the results. In this paper we introduce a new sampling methodology based on cluster sampling that enables the exploitation of ACE analysis to accelerate the fault injection process. In our experiments we demonstrate that state-of-the-art fault injection techniques, generating random faults according to a uniform distribution, is outperformed by the proposed sampling technique, thus enabling several advantages in terms of accuracy and evaluation time. To quantify the introduced benefits we analyzed the micro-architecture reliability of an AMD Southern Islands GPU in presence of single bit upset affecting the vector register file for 6 benchmarks. One of the most important achievements is that considering all the benchmarks, on average, we are one order of magnitude faster/more accurate than uniform-sampling-based techniques in case of non exhaustive fault injection campaigns, while more than two orders of magnitude in case of exhaustive campaigns.**

***Index Terms*—Reliability, GPGPU, fault injection, ACE analysis, cluster sampling.**

I. Introduction

Graphics Processing Units (GPUs) constitute an important part of the recently emerging computing continuum whose total market is more than two billion devices per year and whose application fields range from smartphones to mission-critical data center machines [1]. Technologies of this continuum have introduced benefits for several design parameters (i.e., performance and power consumption) but reliability remains a major concern [2]. Evaluating the reliability of GPU-based systems running complex applications is extremely challenging due to their hardware complexity. This requires complex and time consuming simulations. However, addressing GPUs reliability is necessary since GPUs are finding application in critical scenarios [3]. Accurate and fast techniques able to carefully trade-off between reliability analysis time and accuracy of the reported measurements are required to design complex GPU-based systems and to help reducing power consumption due to reliability over-design [4].

In literature some GPU reliability analysis techniques based on physical error injections have been presented [5], [6]. However, simulation-based techniques are usually preferred in this domain due to the reproducibility of the results and robustness. Similarly to the CPU domain, two reliability evaluation methodologies for GPUs have been established: fault injection [7]–[12] and Architectural Correct Execution (ACE) analysis [10], [12], [13].

A fault injection campaign starts from the generation of a list of faults to be analyzed (injected). For each fault the system is simulated and the fault occurrence is emulated. Eventually, the output of the system is evaluated to understand if the injected fault is masked or a misbehavior is detected. A larger number of injections leads to better accuracy. One of the main drawbacks of this technique is the huge amount of time required to perform simulations to achieve accurate results.

Differently, ACE analysis aims at identifying the hardware resources that need to be correct to have the outcome of the application uncorrupted. This operation is not an easy task, so reliability of the system is always underestimated in order to simplify the analysis and make it faster [12].

This paper presents a methodology to improve reliability analysis of a GPU based system through fault injection. The goal is to improve the evaluation time and accuracy by applying fault pruning techniques coupled with cluster fault sampling techniques.

The idea of pruning the fault list by removing faults whose effect can be determined a priori is not new. In [14] Register Transfer Level (RTL) fault injections into RTL resources (i.e., registers or flip-flops) are limited to those intervals between a write and the last read of the resource. ACE analysis through micro-architecture simulator is a very efficient way to identify these intervals. Similarly, in [7] micro-architecture level fault injection is limited to so called *util resources*, i.e., resources used at least once in the context of the application. When considering the above mentioned pruning techniques, a problem that is often neglected is that their application must be carefully taken into account when sampling the fault list for a fault injection campaign. Other sampling strategies should be adopted instead of the uniform sampling used in [7] to avoid biases and errors in the estimations.

The advantages of the proposed technique with respect to the one described in [7] can be appreciated under two points of view: (i) given a predefined amount of time to perform simulations, obtained results are more accurate, (ii) given a target accuracy, results can be computed faster (i.e., they require less injections). Experiments show that, on average, we are one order of magnitude faster than [7] in case of non exhaustive fault injection campaigns and more than two orders of magnitude faster than [7] in case of exhaustive campaigns.

The paper is organized as follows: Section II describes the proposed methodology and its associated workflow. Section III presents the experimental setup and the results. Finally, Section IV concludes the paper.

II. Methodology

The aim of the paper is the introduction of a methodology to improve fault-injection-based reliability evaluation of an application running on a GPU, in terms of evaluation time and accuracy. For our purpose, we use the Architecture Vulnerability Factor as reliability metric [15]. AVF is the probability that a fault affecting a component manifests at the output of the system (i.e., the result of the application).

As a fault model, in this work we consider single event upsets (SEUs) occurring in the register file of the GPU. These memory elements are of particular interest for saving power as reported in [12]. The proposed approach is based on two steps, i.e., fault pruning and injection sampling described in detail in the following sections.

A. Fault pruning

The goal of fault injection is to establish if a fault corrupts the outcome of the application or, despite its presence, the outcome of the application is still correct. In case of correct output the injected error is masked.

Several fault injection techniques such as the one presented in [9] identify the target faults without taking into account if the outcome of the corresponding injections can be evaluated without simulations. It can happen that some components of the system are unemployed during the execution of an application, thus even if they are affected by a fault, the outcome of the system is correct. Detecting which components and resources are not involved during the computation can save time since injecting a fault into them is not relevant. This is the case of the general purpose register file of a GPU. When a kernel (a GPGPU application) is executed, only a portion of the vector register file is used while the remaining part stays idle. The resources that are used at least once in the context of an application are named util resources [7]. The occupancy, Occ, of a hardware component is defined as the ratio between util resources and the total number of resources. The AVF_{Util} is the AVF computed considering util resources, only. It is related to AVF by the following relation:

$$AVF = AVF_{Util} \times Occ = \frac{\bar{M}}{Inj} = \frac{\bar{M}_{Util}}{Inj_{Util}} \times Occ \quad (1)$$

where $\bar{M}$ is the number of non masked injections considering both the util and non util injections, $\bar{M}_{Util}$ is the number of non masked util injections, Inj is the number of injections considering both the util and non util injections and Inj_{Util} is the number of util injections . Since the non util injections are always masked, then $\bar{M} = \bar{M}_{Util}$. From (1) it can be derived that:

$$Inj_{util} = Inj \times Occ \quad (2)$$

Considering that $0 <= Occ <= 1$, in order to compute the outcome of Inj injections we can just simulate Inj_{Util} injections, thus reducing the number of injections by a factor of $1/Occ$. Resorting to util injections introduces some benefits, however the outcome of many of the util faults can be evaluated without running any simulation too.

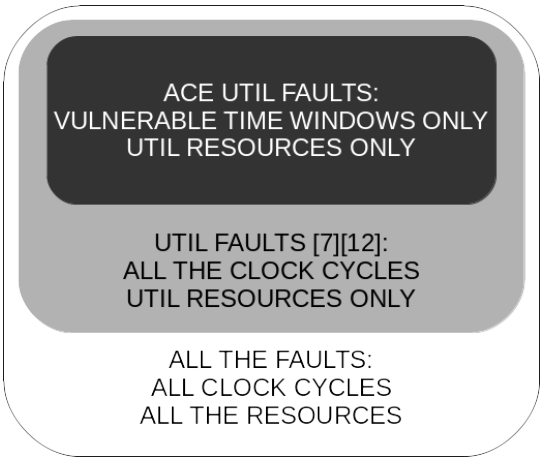


Fig. 1: Faults can affect different resources at different clock cycles.

To further prune the list of candidate faults, we propose to exploit ACE analysis [15]. ACE analysis can be used to identify ACE resources, i.e., resources that must be correct to have an uncorrupted outcome. Particular attention must be paid at the sampling phase when combining ACE analysis and fault injections, as better explained in the next subsection. There are many flavors of ACE analysis and for our purpose we chose the one taking into account read and write sequences of a given hardware resource [15] whose behavior is summarized in Figure 2.

If a fault in a hardware component occurs before a read operation, without being written, the fault is not masked. For this reason a hardware resource is considered as ACE in all time intervals preceding read operations. We name such time intervals as *vulnerable timing windows* (VTW) of a resource.

Combining the concepts of util and ACE resources, fault injections can therefore be limited to *ACE util resources* only, i.e., util resources during their VTW (Figure 1). Similarly to (1):

$$AVF_{Util} = AVF_{ACE\,Util} \times ACE\,Util_{Factor} = \frac{\bar{M}_{Util}}{Inj_{Util}} = \frac{\bar{M}_{ACE\,Util}}{Inj_{ACE\,Util}} \times ACE\,Util_{Factor} \quad (3)$$

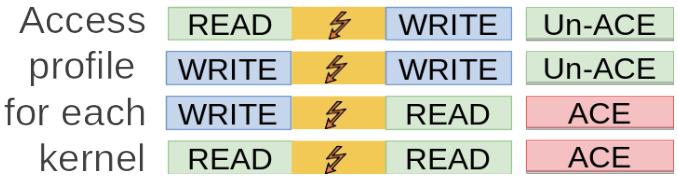


Fig. 2: The vulnerable timing windows considered in our ACE analysis.

where $AVF_{ACE\,Util}$ is the AVF computed considering ACE util injections, that are injections in util resources during their ACE intervals, $\bar{M}_{ACE\,Util}$ is the number of non masked ACE util injections, $Inj_{ACE\,Util}$ is the number of ACE util injections and the $ACEUtil_{Factor}$ is the ratio between the number of all possible ACE util injections and the number of all possible util injections. Applying the same procedure as before, the non ACE util injections are always masked, thus $\bar{M}_{Util} = \bar{M}_{ACE\,Util}$. From (3) it can be derived that:

$$Inj_{ACE\,Util} = Inj_{Util} \times ACEUtil_{Factor} \tag{4}$$

As $0 <= ACEUtil_{Factor} <= 1$, in order to compute the outcome of Inj_{Util} injections we can just simulate $Inj_{ACE\,Util}$ injections, thus reducing the number of injections by a factor of $1/ACEUtil_{Factor}$.

B. Injection sampling

Once all the possible faults are identified, it is required to choose which ones to inject. This phase is named sampling. Usually, in fault injection, faults are sampled uniformly in terms of time and resource since SEUs are assumed to manifest with equal probability of time and location. In details, in a state-of-the-art scenario the addressed faults may affect all possible components at all possible clock cycles with equal probability. When considering util faults only [7], a boundary is applied to the components by selecting a subset of all the possible ones. In this case uniform random sampling is still valid since there is no relation between resources and clock cycles. Util resources are the same for the entire duration of the application. However, when considering ACE util faults a dependency between resources and clock cycles is created: a given resource is ACE util just for given time intervals. As a consequence, uniform random sampling of time and component as employed in fault injection of [7] becomes non optimal for our purposes.

We propose to resort to cluster sampling as a solution. Cluster sampling is based on clusters. Each cluster is a group of individuals. In details we consider a two-stage cluster sampling. This technique consists of two steps: (i) the clusters are sampled and (ii) individuals are sampled from the selected clusters. For our purpose we adapted two approaches based on two-stage cluster sampling from [16]. First step for both of the approaches is the ACE analysis of the executed application required to profile the VTW of all hardware resources. It is worth mentioning that injecting in a VTW results in the computation of the outcome of all the injections points affecting the same resource at a clock cycle belonging to the same VTW. For this reason we can consider each of the VTW as a cluster and its duration expressed in clock cycles (the number of equivalent injections) can be used as a probabilistic weight. Once all the VTW are identified and their weight is quantified, they need to be sampled. Two slightly different approaches have been considered: the first one is the *weight and sample* (WAS), which considers weights at first and later it samples, while the second one is the *sample and weight* (SAW), which samples at first and then it weights.

In the remaining of this section we use the following notation: the number of sampled clusters is n, the number of non-masked ACE util injections is a and the weight is w. Finally the subscript i indicates the reference to the $i-th$ sampled cluster.

- **Weight and sample (WAS)**: the first sampling stage is based on proportional to size sampling (PSS). Clusters are selected with a probability proportional to the associated w_i. Once the clusters are selected an injection is evaluated for each of them. The second sampling stage is based on uniform sampling and the same number of individuals must be analyzed for each of the selected cluster. In this particular case we consider just a single individual per cluster as its outcome is the same to the other individuals in its cluster. If the outcome is non-masked, then $a_i = 1$, otherwise $a_i = 0$. With this approach, adapting the theory introduced in [16] to our case, the $AVF_{ACE\,Util}$ can be estimated as:

$$AVF_{ACE\,Util} = \frac{\sum_{i=1}^{n} a_i}{n} \tag{5}$$

with a standard error equal to:

$$se(AVF_{ACE\,Util}) = \sqrt{\frac{\sum_{i=1}^{n} (a_i - AVF_{ACE\,Util})^2}{n(n-1)}} \tag{6}$$

- **Sample and Weight (SAW)**: the first sampling stage is based on uniform sampling. Clusters are selected with equal probability. Once the clusters are selected an injection is evaluated for each of them. The second stage of the clustering sampling is based on PSS and all the individuals of the selected clusters must be analyzed. In this particular case we consider w_i individuals per cluster. If the outcome of the injection is non-masked, then $a_i = 1$, otherwise $a_i = 0$. In order to guarantee a PSS, a_i is multiplied by the associated w_i. Again, adapting [16] to our case, the $AVF_{ACE\,Util}$ can be computed as:

$$AVF_{ACE\,Util} = \frac{\sum_{i=1}^{n} a_i \times w_i}{\sum_{i=1}^{n} w_i} \tag{7}$$

with a standard error equal to:

$$se(AVF_{ACE\,Util}) \sqrt{\frac{\sum_{i=1}^{n} w_i^2 (a_i - AVF_{ACE\,Util})^2}{n(n-1)\bar{w}^2}} \tag{8}$$

where $\bar{w}$ is the mean w considering all the n clusters.

C. The proposed workflow

The proposed workflow can be applied to micro-architecture and RTL simulators as well. Some changes to the workflow must be adopted with respect to fault injection campaigns described in [7] in order to implement the proposed methodology. In details, we add new steps concerning the fault generation, as reported in Figure 3. First, the application must be profiled to identify the VTW and to acquire information about the execution of the kernels. The VTW are profiled in terms of duration, first clock cycle and the involved architectural general purpose vector register. Information about kernel execution is needed to map architectural registers to the physical memory elements, since this mapping depends on some kernel parameters and it is not the same for all the kernels. After that, the fault pool can be generated according to the chosen cluster sampling technique. Once all the faults are injected and the fault outcomes are classified properly, the final value of the $AVF_{ACE\,Util}$ and $se(AVF_{ACE\,Util})$ can be computed respectively according to (5) and (6) for *WAS*, and to (7) and (8) for *SAW*.

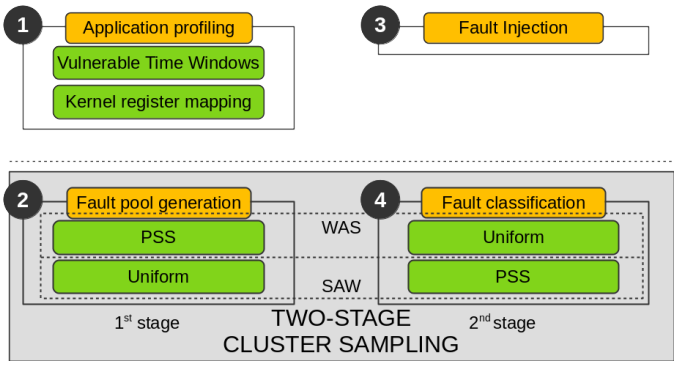


Fig. 3: The workflow: first step is the profiling, second step is the fault pool generation, third step is fault injection, final step is fault classification and evaluation.

III. Experimental Results

A. Experimental setup

For our purpose we implemented the workflow described in subsection II-C in the multi2sim micro-architecture simulator [17]. The target GPU is the AMD HD Radeon™7970 belonging to the Southern Islands architecture. We performed the reliability analysis for 6 AMD-APP SDK benchmarks[1]: Bitonic Sort (BS), Discrete Cosine Transform (DCT), Matrix Transpose (MT), Reduction (RED), Scan Large Arrays (SLA) and Simple Convolution (SC). For each benchmark we identified all VTW by profiling the involved resource, the start cycle and the duration. The amount of information required per benchmark is proportional to column VTW of Table I. We ran a fault simulation for each of VTW in order to compute the exact value of the AVF_{Util} as described in Section II.

[1]AMD-APP-SDK v.2.5 for multi2sim available at: https://github.com/Multi2Sim/m2s-bench-amdsdk-2.5

TABLE I: The number of util faults is very big and an exhaustive fault injection campaign requires huge duration. Resorting to ace faults introduces marginal benefits, while addressing *vulnerable time windows* is the best option being from 2 to 3 orders of magnitude faster.

Benchmark	UTIL	ACE	VTW	INJ RED
BS	415841280	32939648	731136	569x
DCT	3488235520	905388032	7274496	480x
MT	845152256	408547328	4063232	208x
RED	1532344320	362733120	1447968	1058x
SLA	5531212800	693824864	2360736	2343x
SC	534282240	455500864	3248512	164x
AVG	2057844736	476488976	3187680	804x

The goal of the experimental campaign is to compare the accuracy of the estimated AVF_{Util} when using the proposed approaches (i.e., *WAS*, *SAW*) with those obtained by applying the technique proposed in [7]. Reliability analysis is computed for different number of samples ranging from 10 to 6 millions. For each number of samples, each technique and each application, the reliability evaluation is repeated 100 times so that we were able express the difference between the obtained results and the exact ones in terms of Mean Squared Error (MSE).

B. Results

One of the goals of our experiments is to demonstrate how the fault injection campaigns can be pruned. Trying to identify *a priori* faults that are masked is a valuable method, introducing remarkable benefits as shown in Figure 4, which reports the proportion of the faults distinguishing among non-util (NU) faults, util non-ace (UNA) faults and ace util faults (A). It can be noticed that reduction of the fault space from NU to util (UNA and A) is of several orders of magnitude. On the opposite, the reduction form util (UNA and A) to ace faults is not so big, it is marginal for some benchmarks as MT and SC.

Table I illustrates the number of all possible util faults, ACE faults and VTW. The ratio between util and ACE faults is not as high as the ratio between ACE faults and VTW. This implies that addressing ACE faults only would introduce just marginal advantages. Instead, dealing with VTW allows us to reduce the number of injections with respect to util faults of at least 164 times (for SC). Such a reduction strongly depends on the application: in the best case it is equal to 2,343 times (SLA), while on average it is 804 times.

The MSE of each benchmark varying the number of samples and the sampling technique is reported in Figure 5, alongside the exact value of the AVF_{Util} computed in exhaustive fault injection campaigns. The convergence curve of all the techniques is very similar and it is proportional to $\sqrt{n}$, where n is the number of samples, as expected.

In all the cases SAW technique outperforms the others. The maximum difference between SAW and the other happens for SLA where it reaches more than one order of magnitude, while it is negligible for SC if compared to [7]. More specifically, the gap between SAW and [7] seems to be correlated to the number of UNA faults shown in Figure 4. This is verified for

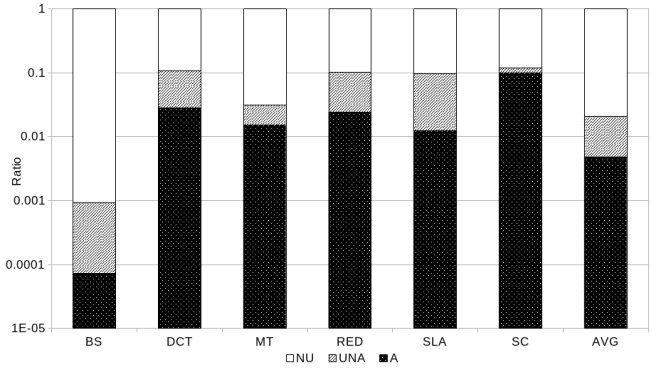


Fig. 4: Most of the faults can be classified as masked without running any simulation. This is the case of non util (NU) and util non ace (UNA), representing more than 99% of all the faults on average. This proves that moving towards better techniques to reduce the number of fault simulations would enable fast results of exhaustive fault injection campaigns.

SC and MT which have the smallest percentage of UNA faults and exhibit the smallest gap. Differently SLA and BS have the largest percentage of UNA faults and exhibit the largest gap.

SAW technique has a less predictable behavior as for BS and RED the position of its MSE is in between SAW and [7], for MT and SLA its MSE is very similar to [7], while for SC and DCT it has the worst MSE. This suggests that the MSE of SAW strongly strongly depends on the application. In fact, (8) gives a valuable explanation: the error depends on the distribution of the VTW.

Finally, observing all the graphs of Figure 5, it can be stated that some benchmarks converge faster than others. BS is the fastet, SC is the slowest, while DCT, MT, RED and SLA are in the middle. This trend is also present when analyzing the percentage of ace faults (A) in Figure 4. In fact, ace faults represent our sampling space and smaller sampling spaces imply faster convergence rate.

IV. Conclusion

This paper introduces a new methodology for the sampling and the computation of the results based on cluster sampling. Thanks to this sampling strategy ACE analysis can be exploited to limit the resources to analyze through fault injection. We adapted the two cluster sampling techniques presented in [16] which best fit our purpose. Experiments demonstrate that the WAS is the best strategy, outperforming both WAS and [7] in all the analyzed benchmarks. The proposed solution is on average one order of magnitude faster/more accurate in case of non-exhaustive fault injection campaign and two orders of magnitude for exhaustive campaigns, allowing to compute the exact value of AVF_{Util} for some small applications running on a GPU for the first time.

References

[1] D. Buchholz and I. J. Dunlop, "The future of enterprise computing: Preparing for the compute continuum," *IT@ Intel White Paper, Intel IT*, 2011.

[2] A. Biswas, "Cost-effective reliability trade-offs and challenges," April 2018.

[3] P. Rech, L. Pilla, P. Navaux, and L. Carro, "Impact of GPUs Parallelism Management on Safety-Critical and HPC Applications Reliability," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, Jun. 2014, pp. 455–466.

[4] G.-H. Asadi, V. Mehdi, B. Tahoori, and D. Kaeli, "Balancing Performance and Reliability in the Memory Hierarchy," in *IEEE International Symposium on Performance Analysis of Systems and Software, 2005. ISPASS 2005.* IEEE, Mar. 2005, pp. 269–279.

[5] P. Rech, C. Aguiar, R. Ferreira, M. Silvestri, A. Griffoni, C. Frost, and L. Carro, "Neutron-induced soft errors in graphic processing units," in *2012 IEEE Radiation Effects Data Workshop*, July 2012, pp. 1–6.

[6] I. S. Haque and V. S. Pande, "Hard data on soft errors: A large-scale assessment of real-world error rates in gpgpu," in *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, May 2010, pp. 691–696.

[7] N. Farazmand, R. Ubal, and D. Kaeli, "Statistical fault injection-based avf analysis of a gpu architecture," in *Proceedings of the IEEE Workshop on Silicon Errors in Logic - System Effects*, 2012.

[8] B. Fang, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi, "GPU-Qin: A methodology for evaluating the error resilience of GPGPU applications," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, Mar. 2014, pp. 221–230.

[9] S. Tselonis and D. Gizopoulos, "GUFI: A framework for GPUs reliability assessment," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, Apr. 2016, pp. 90–100.

[10] A. Vallero, D. Gizopoulos, and S. Di Carlo, "SIFI: AMD southern islands GPU microarchitectural level fault injector," in *2017 IEEE 23rd International Symposium on On-Line Testing and Robust System Design (IOLTS)*. IEEE, Jul. 2017, pp. 138–144.

[11] S. K. S. Hari, T. Tsai, M. Stephenson, S. W. Keckler, and J. Emer, "SASSIFI: An architecture-level fault injection tool for GPU application resilience evaluation," in *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, Apr. 2017, pp. 249–258.

[12] A. Vallero, S. Tselonis, D. Gizopoulos, and S. Di Carlo, "Multi-faceted microarchitecture level reliability characterization for NVIDIA and AMD GPUs," in *2018 IEEE 36th VLSI Test Symposium (VTS)*. IEEE, Apr. 2018, pp. 1–6.

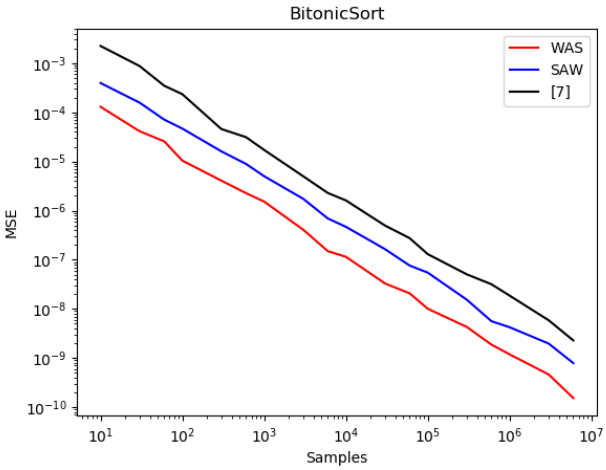
[13] J. Tan, N. Goswami, T. Li, and X. Fu, "Analyzing soft-error vulnerability on GPGPU microarchitecture," in *2011 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, Nov. 2011, pp. 226–235.

[14] A. Benso, M. Rebaudengo, L. Impagliazzo, and P. Marmo, "Fault-list collapsing for fault-injection experiments," in *Annual Reliability and Maintainability Symposium. 1998 Proceedings. International Symposium on Product Quality and Integrity*, Jan 1998, pp. 383–388.

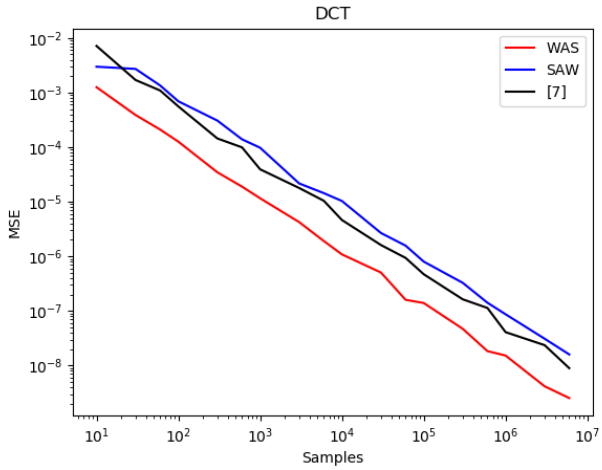
[15] S. Mukherjee, C. Weaver, J. Emer, S. Reinhardt, and T. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," in *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.* IEEE, Dec. 2003, pp. 29–40.

[16] R. Frerichs, "Cluster sampling, chapter five," *Rapid Surveys*, 2004.

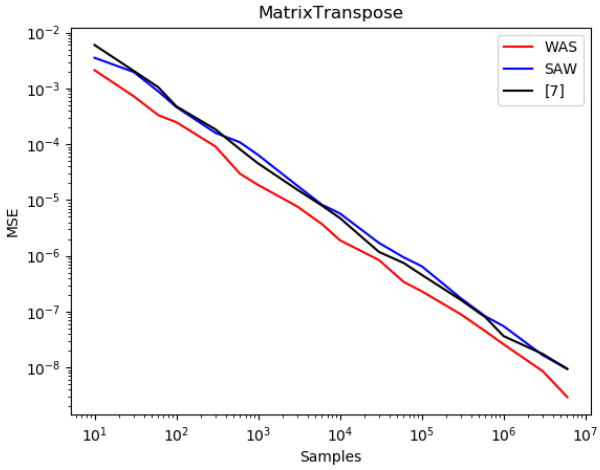
[17] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, "Multi2sim: a simulation framework for cpu-gpu computing," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*. ACM, 2012, pp. 335–344.



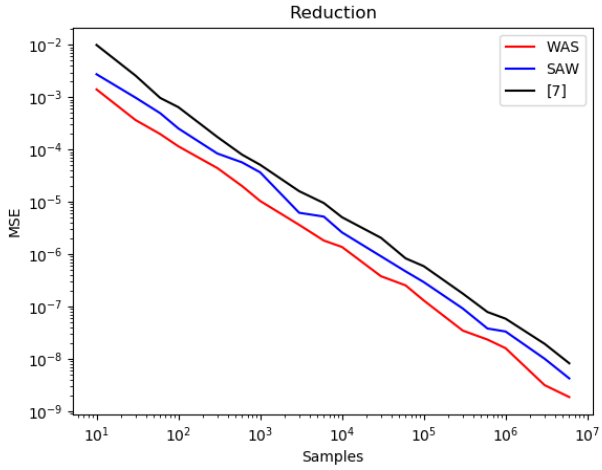
(a) The MSE for the BS, where $\bar{Y} = 0.056491$



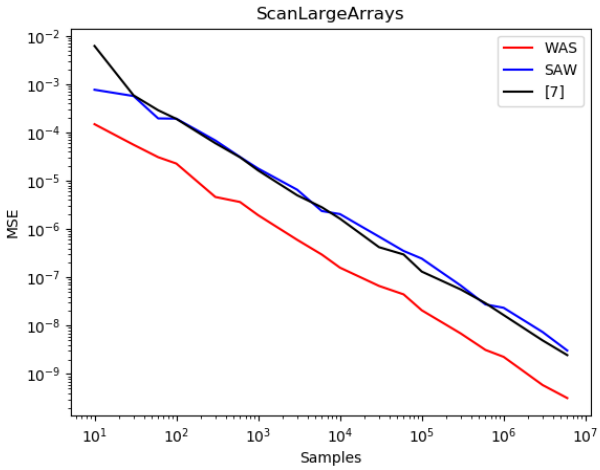
(b) The MSE for the DCT, where $\bar{Y} = 0.194105$



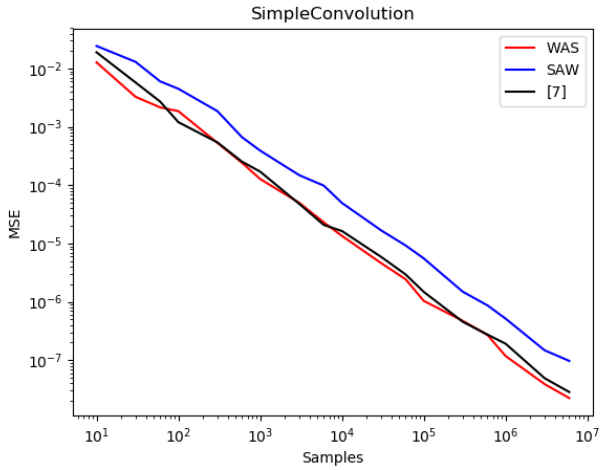
(c) The MSE for the MT, where $\bar{Y} = 0.429848$



(d) The MSE for the RED, where $\bar{Y} = 0.150909$



(e) The MSE for the SLA, where $\bar{Y} = 0.107056$



(f) The MSE for the SC, where $\bar{Y} = 0.630058$

Fig. 5: The MSE is computed for each benchmark evaluating the AVF_{Util} for each combination of number of samples and technique. For each combination the AVF_{Util} is estimated 100 (n) times. The $MSE = (\sum_i^n (Y_i - \bar{Y})^2)/n$
, where Y_i is one evaluation of the AVF_{Util} and $\bar{Y}$ is the exact value of the AVF_{Util} computed through the exhaustive fault injection campaigns. $\bar{Y}$ is also reported for each benchmark.