CAROL-FI: an Efficient Fault-Injection Tool for Vulnerability Evaluation of Modern HPC Parallel Accelerators

*Daniel Oliveira, *Vinicius Fratin, *Philippe Navaux, †Israel Koren, *Paolo Rech
*Institute of Informatics †University of Massachusetts
UFRGS UMass
Porto Alegre, Brazil Amherst, US

***Abstract*—Transient faults are a major problem for large scale HPC systems, and the mitigation of adverse fault effects need to be highly efficient as we approach exascale. We developed a fault injection tool (CAROL-FI) to identify the potential sources of adverse fault effects. With a deeper understanding of such effects, we provide useful insights to design efficient mitigation techniques, like selective hardening of critical portions of the code.**

We performed a fault injection campaign injecting more than 67,000 faults into an Intel Xeon Phi executing six representative HPC programs. We show that selective hardening can be successfully applied to *DGEMM* and *Hotspot* while *LavaMD* and *NW* may require a complete code hardening.

I. Introduction

Accelerators are nowadays extensively used to expedite calculations in large HPC centers. Intel *Xeon Phi* devices and NVIDIA GPUs, for instance, are employed in 5 of the top 10 supercomputers [5]. The main reason to use accelerators are their high computational capacity, low cost, reduced energy consumption, and flexible development platforms.

Unfortunately, as accelerators are also extremely likely to experience transient errors as they are built with cutting-edge technology, have very high operation frequencies, and include large amounts of memory and logic resources. The generated error may corrupt data values or logic operations and lead to Silent Data Corruption (SDC), Detected Uncorrectable Error (DUE), or be masked and cause no observable error [14]. As a reference, DOE's Titan, today's third most powerful supercomputer [5] composed of more than $18,000$ $Kepler$ GPUs, has a radiation-induced Mean Time Between Failures (MTBF) in the order of dozens of hours [16]. As we approach exascale, the resilience challenge will become even more critical due to an increase in system scale [12].

In this paper, we present a detailed analysis of transient errors vulnerability using an in-house high-level fault injector tool, named CAROL-FI. Unlike most fault-injection frameworks, CAROL-FI injections are made at the highest possible level. CAROL-FI is intended as a tool to help developers to identify the portions of their code that, once corrupted, are more likely to affect the output and can then provide pragmatic information to move towards a solution of the reliability issue in HPC. The distinction between critical and not critical portions of the benchmark, in fact, allows to selectively harden only a subset of instructions or variables. As a result, this can significantly improve code reliability while introducing minimal overhead.

In summary, this paper makes the following contributions:

- We present an in-house high-level fault injection tool for Intel Xeon Phi, which is made publicly available [1].
- We define a methodology to identify critical portions of HPC benchmarks and include a discussion on possible mitigation techniques.

The rest of this paper is organized as follow. Section II provides a brief background on the impact of transient faults in HPC and presents related work. Section III describes CAROL-FI, our in-house fault injector tool. Section IV describes the methodology used in this work. Section V presents the evaluation of six HPC benchmarks based on the fault injection campaign. Finally, Section VI concludes the paper and proposes future work.

II. Background

A. Transient Errors Effects

The pursuit of extreme performance coupled with reduction of power consumption and an increase in computing resources makes modern HPC computing devices extremely prone to transient errors. The main causes of transient errors at the physical level are voltage-frequency variations, temperature perturbations, and electromagnetic interferences. Lately, neutron-induced transient errors have been shown to be particularly critical for HPC systems [4], [16].

B. Existing Software Fault Injection Tools

One of the most common approaches to evaluate the reliability of a component or an application is to use software fault injection. By injecting a fault it is possible to calculate the Architectural Vulnerability Factor (AVF), which is the probability for a corruption to propagate to the output [13], or the Program Vulnerability Factor (PVF), which is the probability that a fault at the instruction level will affect the program output [15]. The high-level fault injector that we have developed provides information about the most vulnerable parts of a given code and about the effect of a corrupted variable on the code output. While high-level fault injection can not provide realistic error rates, it can identify the critical portions of the code.

There are several available software fault injection tools that differ in terms of injection methods and domains of application. Some examples can be found in [9], [7], [17], [10], [11]. However, There are still no available fault injection tools capable of injecting faults into x86 parallel devices (e.g., Intel Xeon Phi) and, at the same time, provide information about the error propagation paths to the application level (i.e., the variable and code fragment that generated an error).

III. CAROL-FI

Our fault injector, CAROL-FI (publicly available at [1]) is built upon GDB (the GNU debugger) with Python support. The goal of CAROL-FI is not estimating the realistic error rate, but obtain information about the error propagation paths and provide useful insights to the code designer on how to mitigate their effects. Compiling the code in debug mode allows to gather this information. Thus, Debug information is used to correlate each allocated memory portion with its corresponding variable in the source code.

CAROL-FI's workflow consists of the following steps:

1) The evaluated code is launched by GDB.
2) At a random time an interrupt signal is sent to the program, which stops its execution and the GDB triggers a python script that examines and change memory content.
3) The python script navigates the execution stack of the program to randomly choose a variable. The script can navigate the current frame up to the main frame of the program. Thus, variables from the current to the main scope can be chosen.
4) Based on the fault model one or more bits used by the selected variable are flipped.
5) After the fault is injected, GDB resumes the execution of the program.

The current fault model is based on a single bit-flip of a single variable. However, the fault model can be easily extended to simulate multiple bit-flips or any other fault model of interest.

CAROl-FI is quite fast as its only significant overheads are the ones caused by the GDB and the debug mode that disables compiler optimizations. On the average, its overhead is about $4x$ the normal execution time, with a worst case of $8x$.

It is important to note that, while we inject faults in variables, fault injection in the code data is emulating much more than main memory errors. Errors in the cache, and even in logic circuits will eventually propagate to the main memory. Thus, main memory corruption can be the result of faults in the cache, registers, instructions, or transient spikes in logic circuits. Therefore, even a memory with an appropriate error protection mechanism will not protect the component from all the harmful effects induced by CAROL-FI.

IV. Methodology

The Xeon Phi board used for evaluating CAROL-FI capabilities and analyze HPC codes vulnerabilities is the coprocessor 3120A, codenamed Knights Corner. The 3120A coprocessor has 57 physical in-order cores, and each one has 32 512-wide vector registers and supports four hardware threads. The Operating System is the CentOS 7.0 with Intel MPSS version 3.7 and GDB 7.8 with Intel extensions.

To evaluate the capabilities of CAROL-FI, we selected six benchmarks from different domains with different computation and communication patterns. The selected benchmarks are: a **Matrix Multiplication (DGEMM)** benchmark, a DOE mini-app named **CLAMR** [8], and **HotSpot**, **LavaMD**, **LUD** and **Needleman-Wunsch (NW)** which are mini-apps from the Rodinia benchmark suite [2].

We have injected at least 10,000 faults into each of the selected benchmarks. For every fault injection experiment, we compared the output of the program execution to a previously computed golden copy. CAROL-FI also kill the program if a user-defined time limit is surpassed, we set the time limit to be twice the program's execution time on average.

Four possible outcomes can be observed using CAROL-FI: **Masked**, **potentially Masked**, **SDC**, and **DUE**. The potentially Masked classification is important for applications that can accept small deviations from the correct result. For instance, a geophysics application [3] tolerates up to 4% of deviations. Moreover, this concept of acceptable error is intrinsic of imprecise computation [6] where a certain deviation from the correct result is acceptable. Therefore, we choose to consider as potentially masked executions with a deviation up to 2%.

V. Reliability and Criticality Evaluation

Figure 1 presents the percentage of faults that are masked, potentially masked, or cause an SDC or DUE for each of the six benchmarks presented in Section IV. For most of the benchmarks, SDCs are less likely to occur than DUEs, while the majority of injected faults are masked during computation (except DGEMM).

As shown in Figure 1, about 60% of the faults injected in DGEMM generate an error (SDC or DUE). Most of the SDCs and DUEs are the result of faults injected in the input and output **matrices** and **loop control variables**. Analyzing the outcomes of injections in the matrices we find that on average only 25% of injections generate an SDC, while 20% produce a potentially masked error and 19% cause a DUE. When analyzing the vulnerability of loop control variables we discovered that 43% of the faults injected in those variables generate an SDC and 54% cause a DUE. DGEMM creates only nine loop control variables of integer type. However, each of the 228 threads active in parallel on the Xeon Phi allocates those nine integers to have its own copy of the loop control variables, resulting in a significant memory portion used to store loop control variables. As a result, the probability of having a corrupted loop control variable becomes significant, and the criticality of that corruption is very high.

CLAMR results show that 75% of the injected faults do not generate an observable error, as shown in Figure 1. We have discovered that faults injected in the mesh code produce the vast majority of errors. We can divide the mesh operations into three parts: **Sort**, **Tree**, and **others**. Of all the injections in

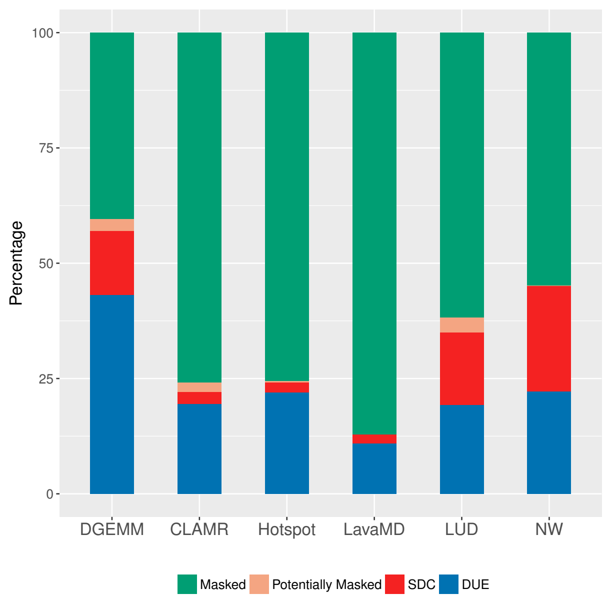

Fig. 1: Fault injection campaign results per benchmark for Intel Xeon Phi.

the Sort part of CLAMR, 15% generate an SDC, 12% produce potentially masked mismatches, and 40% causes DUE. For the Tree part, only 15% of all the Tree fault injections generate an SDC, 1% produce potentially masked, and 40% cause a DUE effect. All the faults in the remaining variables of the mesh code are classified as others. Only 8% of the faults in the other code generate an SDC, 23% produce mismatches that can be classified as potentially masked, and 51% cause DUE. The fault injection analysis shows that Mesh operations and structure are the most sensitive. Furthermore, Sort and Tree operations are equally sensitive, causing the majority of the harmful outcomes.

Hotspot fault injections show a similar trend to that observed for CLAMR. 75% of the faults are masked and do not affect the output. As we can see in Figure 1, the percentage of faults that generate SDCs is low, being less than 22%. Most of the observed SDCs are caused by faults in **constant variables** used during computation. Our fault injection analysis shows that less than 10% of faults in the Hotspot constant variables cause an SDC. DUE are caused most of the times by two **control variables** that store the number of the grid rows and columns. The probability of a fault in a grid control variable to cause a DUE is extremely high, about 96%. Hotspot computes the temperature of functional blocks in a chip given the power consumed by these blocks. The temperatures of the different blocks are calculated in an iterative manner, and errors in intermediate values can be dissipated out of the system while the solution is converging. Thus, Hotspot is intrinsically robust to data errors.

Figure 1 shows that for LavaMD, only 15% of the injected faults produce SDC or DUE. Faults in the **input arrays** and **control variables** cause the vast majority of harmful outcomes. The *charge* and *distance* arrays used as inputs to the algorithm are responsible for 25% of the observed SDCs and 10% of the DUEs. The two arrays are up to 5 orders of magnitude bigger than the other data structures that cause harmful effects. Thus, the probability of a fault to occur in the two input arrays is larger than for the other data structures. Therefore, The two arrays are the most critical part of the algorithm.

LUD exhibits a behavior similar to that of DGEMM. Most of the harmful outcomes are due to faults in the **matrices** and **control variables**. However, the DUE and SDC rate for LUD are well balanced, LUD has a much lower DUE rate than DGEMM (see Figure 1). Faults in the main matrix and the temporary matrices allocated during the computation of the decomposition generate an SDC for about 40% of the faults injected into them, and about 30% cause potentially Masked outcomes. Evaluating the control variables, we observe that 20% of the faults generate an SDC and 40% cause a DUE. We notice a clear distinction between SDCs and DUEs as faults in the matrices generate most of the SDCs, and faults in the control variables are responsible for most DUE.

NW has the most well-balanced rate between SDC and DUE, as Figure 1 shows. The rates of SDCs and DUEs are similar because faults that cause the vast majority of errors are in the **matrices** used as input and output. We notice that 49% of the injected faults in the matrices generate an SDC, and 49% cause a DUE.

A. Discussion

CAROL-FI results provide in-depth insights into the sources of errors. For instance, for DGEMM we find that protecting all the matrices data is ineffective, but protecting the loop control variables will greatly benefit the code reliability. In CLAMR, we can partition the origins of harmful effects into two basic operations from the mesh code (Sort and Tree), and find that both are equally responsible for SDC and DUE. Thus, we can focus on the protection of these operations. Hotspot is naturally robust, and we should protect only the small portion of data that can cause harm, saving considerable amounts of execution time and energy. LavaMD and NW present the biggest challenges, showing that partial protection may prove ineffective due to the fact that a significant portion of the data is critical. Finally, LUD shows a clear distinction between the types of harmful effects and the sources of such effects. Then, to mitigate one of the effects we can target only selected data structures.

CAROL-FI can be used to assess the vulnerability of each structure in the source code, and to identify the type of harmful effect that a data structure may produce. This information is essential when attempting to mitigate the impact of transient errors in commercial off-the-shelf components such as consumer server processors and accelerators like Intel Xeon Phi. As the user is unable to modify the hardware and make it less sensitive to faults, understanding the potential impact of faults at a high level of the code is essential to allow producing a reliable code in today computing systems. Moreover, our fault injector can provides general hints, such as the use of partial ECC, to system designers if they wish to design more reliable hardware that adapts to the user needs.

VI. Conclusion and Future Work

In this paper we have presented our fault injector tool, CAROL-FI, and provided an analysis of the vulnerabilities of several HPC programs. Transient faults are one of the main concerns for today's and future HPC systems. Our tool is capable of detecting the high-level sources of harmful effects and of providing useful insights for mitigating them. With the insight provided, architecture designers can build more reliable hardware, and HPC programmers can devise smart mitigation techniques.

In the future, we will improve further CAROL-FI to collect more useful data. The improved fault injector will provide a deeper understanding of the code, such as the dependence of the impact of faults on the timing of their occurrence. We also plan to implement the mitigation techniques proposed and validate them with fault injection campaigns and radiation beam experiments.

Acknowledgments

This work received partial funding from CAPES/PVE, the EU H2020 Programme, and from MCTI/RNP-Brazil under the HPC4E project, grant agreement n° 689772.

References

[1] "Carol-fi." https://github.com/UFRGS-CAROL/carol-fi, 2017.

[2] 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 *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, Oct. 2009, pp. 44–54.

[3] J. de la Puente, M. Ferrer, M. Hanzich, J. E. Castillo, and J. M. Cela, "Mimetic seismic wave modeling including topography on deformed staggered grids," *GEOPHYSICS*, vol. 79, no. 3, pp. T125–T141, 2014.

[4] D. A. G. de Oliveira, L. L. Pilla, T. Santini, and P. Rech, "Evaluation and mitigation of radiation-induced soft errors in graphics processing units," *IEEE Transactions on Computers*, vol. 65, no. 3, pp. 791–804, March 2016.

[5] J. Dongarra, H. Meuer, and E. Strohmaier, "TOP500 Supercomputer Sites: June 2016," 2016. [Online]. Available: http://www.top500.org

[6] D. Ernst, S. Das, S. Lee, D. Blaauw, T. Austin, T. Mudge, N. S. Kim, and K. Flautner, "Razor: circuit-level correction of timing errors for low-power operation," *IEEE Micro*, vol. 24, no. 6, pp. 10–20, Nov 2004.

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

[8] Q. Guan, N. DeBardeleben, B. Artkinson, R. Robey, and W. Jones, "Towards Building Resilient Scientific Applications: Resilience Analysis on the Impact of Soft Error and Transient Error Tolerance with the CLAMR Hydrodynamics Mini-App," in *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*, Sept 2015, pp. 176–179.

[9] S. K. S. Hari, T. Tsai, M. Stephenson, S. W. Keckler, and J. Emer, "Sassifi: Evaluating resilience of gpu applications," in *Proceedings of the Workshop on Silicon Errors in Logic-System Effects (SELSE)*, 2015.

[10] D. Li, J. S. Vetter, and W. Yu, "Classifying soft error vulnerabilities in extreme-scale scientific applications using a binary instrumentation tool," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 57:1–57:11.

[11] G. Li, K. Pattabiraman, C.-Y. Cher, and P. Bose, "Understanding error propagation in gpgpu applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 21:1–21:12. [Online]. Available: http://dl.acm.org/citation.cfm?id=3014904.3014932

[12] R. Lucas, "Top ten exascale research challenges," in *DOE ASCAC Subcommittee Report*, 2014.

[13] S. S. Mukherjee *et al.*, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," in *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2003.

[14] 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, Nov 2005.

[15] V. Sridharan and D. R. Kaeli, "Eliminating microarchitectural dependency from architectural vulnerability," in *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, Feb 2009, pp. 117–128.

[16] D. Tiwari, S. Gupta, J. Rogers, D. Maxwell, P. Rech, S. Vazhkudai, D. Oliveira, D. Londo, N. Debardeleben, P. Navaux, L. Carro, and A. B. Bland, "Understanding GPU Errors on Large-scale HPC Systems and the Implications for System Design and Operation," in *Proceedings of 21st IEEE Symp. on High Performance Computer Architecture (HPCA)*. ACM, 2015.

[17] 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)*, April 2016, pp. 90–100.