Experimental and Analytical Study of Xeon Phi Reliability

Daniel Oliveira
Institute of Informatics, UFRGS
Porto Alegre, RS, Brazil

Laércio Pilla
Department of Informatics and Statistics, UFSC
Florianópolis, SC, Brazil

Nathan DeBardeleben
Los Alamos National Laboratory
Los Alamos, NM, US

Sean Blanchard
Los Alamos National Laboratory
Los Alamos, NM, US

Heather Quinn
Los Alamos National Laboratory
Los Alamos, NM, US

Israel Koren
University of Massachusetts, UMass
Amherst, MA, US

Philippe Navaux
Institute of Informatics, UFRGS
Porto Alegre, RS, Brazil

Paolo Rech
Institute of Informatics, UFRGS
Porto Alegre, RS, Brazil

ABSTRACT

We present an in-depth analysis of transient faults effects on HPC applications in Intel Xeon Phi processors based on radiation experiments and high-level fault injection. Besides measuring the realistic error rates of Xeon Phi, we quantify Silent Data Corruption (SDCs) by correlating the distribution of corrupted elements in the output to the application's characteristics. We evaluate the benefits of imprecise computing for reducing the programs' error rate. For example, for HotSpot a 0.5% tolerance in the output value reduces the error rate by 85%.

We inject different fault models to analyze the sensitivity of given applications. We show that portions of applications can be graded by different criticalities. For example, faults occurring in the middle of LUD execution, or in the Sort and Tree portions of CLAMR, are more critical than the remaining portions. Mitigation techniques can then be relaxed or hardened based on the criticality of the particular portions.

CCS CONCEPTS

• **Computer systems organization → Parallel architectures**;
• **Hardware → Fault tolerance**;

KEYWORDS

Radiation experiments, Fault Injection, Parallel Architectures, Fault Tolerance, Reliability

ACM Reference Format:
Daniel Oliveira, Laércio Pilla, Nathan DeBardeleben, Sean Blanchard, Heather Quinn, Israel Koren, Philippe Navaux, and Paolo Rech. 2017. Experimental and Analytical Study of Xeon Phi Reliability. In *Proceedings of SC17*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3126908.3126960

SC17, November 12–17, 2017, Denver, CO, USA

ACM ISBN 978-1-4503-5114-0/17/11…$15.00
https://doi.org/10.1145/3126908.3126960

1 INTRODUCTION

Accelerators are extensively used to expedite calculations in large HPC centers. Tianhe-2, Cori, Trinity, and Oakforest-PACS use Intel *Xeon Phi* and many other top supercomputers use other forms of accelerators [17]. The main reasons to use accelerators are their high computational capacity, low cost, reduced per-task energy consumption, and flexible development platforms. Unfortunately, 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 resources.

Reliability has been identified by the U.S. Department of Energy (DOE) as one of the ten major challenges for exascale [42]. Errors that may undermine the reliability of an HPC system can come from a variety of sources including environmental perturbations, firmware errors, manufacturing process, temperature, and voltage variations [31, 35, 37]. Such errors 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 [13, 43, 44]. Radiation-induced soft errors are particularly critical, as they have been found to dominate error rates in commercial devices [4]. The large scale and long application executions in leading scientific HPC centers exacerbate the probability of having a transient error in the system. As a reference, DOE's Titan, composed of more than 18,000 *Kepler* GPUs, has a radiation-induced Mean Time Between Failures (MTBF) in the order of dozens of hours [21, 46]. As we approach exascale, the resilience challenge will become even more critical due to an increase in system scale [34, 42, 45]. In this scenario, a lack of understanding of HPC device resilience may lead to lower scientific productivity and significant monetary loss [45].

Our intention is to *evaluate*, *understand*, and *develop mitigation strategies* for reliability issues in current and future supercomputers. First, we *evaluate* the problem by showing the results of our neutron beam experiments on *Knights Corner* Xeon Phi (3120A) components. We report the realistic SDC and DUE Failure In Time (FIT) rates of five benchmarks. Each benchmark was tested for more than 100 hours at the Los Alamos Neutron Science Center (LANSCE), providing data that covers more than 57,000 years of natural exposure per board. All the collected errors are available on a public repository to allow third party analysis and to ensure reproducibility [40].

Depending on the application and circumstances, some SDCs that are satisfactory close to the correct results may be tolerated in HPC [7, 14]. In order to consider the outputs' error severity and better *understand* the reliability issue, our evaluation considers how errors manifest at the benchmark output and measures how the Xeon Phi FIT rate reduces as a function of the tolerated level of imprecision in the output. For *HotSpot*, for instance, the FIT rate is reduced by 85% if a 0.5% variation in the output value is acceptable.

We then performed a detailed analysis of the applications' vulnerabilities to transient errors using a high-level fault injector tool, named CAROL-FI. Some fault injectors, like LLFI, GPU-Qin, SASSIFI, and GUFI are already available for processors and GPUs [3, 18, 23, 33, 47]. Unlike most fault-injection frameworks, CAROL-FI injections are made at the highest possible level, to identify the benchmark portions that are more likely to generate an SDC or a DUE. 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 *develop mitigation strategies* for the reliability issue in HPC. The distinction between critical and not critical portions of the benchmark, in fact, allows one to selectively harden only a subset of instructions or variables. As a result, this can significantly improve code reliability while introducing minimal overhead.

To demonstrate the utilization and benefits of the proposed fault injection tool, we analyzed fault injection results for six different benchmarks. We report data obtained by injecting more than 90,000 transient faults, identifying the selected benchmarks' critical parts.

In summary, this paper makes the following contributions:

- A neutron beam-based evaluation of Xeon Phi's FIT rates, which includes a collection of publicly available neutron-induced output errors.
- We qualify SDCs by considering their spatial distribution and the benefits of imprecise computation for HPC reliability.
- We present an in-house high-level fault injection tool for Intel Xeon Phi, which is made publicly available [9].
- Through an analytical methodology we identify critical portions of HPC benchmarks and include a discussion on possible mitigation techniques.

The rest of this paper is organized as follows. Section 2 gives a brief background on the impact of transient faults in HPC and presents related work. Section 3 describes the methodology used in this work. Section 4 presents the neutron beam experiments. Section 5 describes CAROL-FI, our in-house fault injection tool. Section 6 presents the evaluation of six HPC benchmarks based on the fault injection campaign. Finally, Section 7 concludes the paper.

2 BACKGROUND

2.1 Transient Errors Effects in HPC

The pursuit of extreme performance coupled with increased power efficiency and an increase in computing resources makes modern HPC computing devices extremely prone to transient errors. The main causes of transient errors are voltage-frequency variations, temperature perturbations, and electromagnetic interferences. Lately, neutron-induced errors have been shown to be critical for HPC systems [15, 46]. A flux of about 13 $neutrons/((cm^2) \times h)$ reaches ground at sea level, and the flux exponentially increases with altitude [29]. A neutron strike may perturb the transistor state, generating bitflips in memory or a current spike in logic circuits that, if latched, leads to a fault [8, 36].

A transient error (independent of its source) leads to one of the following outcomes: (1) no effect on the program output (the failure is **masked** or corrupted data is not used), (2) Silent Data Corruption (**SDC**), i.e., incorrect program output, or (3) Detected Unrecoverable Error (**DUE**), which typically is a program crash or device reboot. Highly parallel computing architectures, like the Xeon Phi, have some reliability weaknesses [15, 16, 21, 49]. For instance, a single particle generating a radiation-induced failure in the scheduler or shared memories (used to expedite parallel executions), is likely to affect the computation of several parallel threads. Additionally, a single corrupted thread could feed various other threads with erroneous data, again leading to multiple corrupted elements. Our experiments on the Xeon Phi, shown in Section 4.3, demonstrate that less than 10% of neutron-corrupted executions are affected by only a single erroneous element in the output. Finally, while HPC accelerators have the main storage structures protected with Error-Correcting Code (ECC) implementing Single Error Correction Double Error Detection (SECDED), some major resources are left unprotected, such as flip-flops in pipelines queues, logic gates, instruction dispatch units, and interconnect network. As experimentally demonstrated in Section 4, the neutron-induced error rate of Xeon Phi can be as high as 193 FIT, even if ECC is enabled.

SDCs are not always critical for HPC applications but some output errors can be tolerated. For instance, if the corruption affects only the least significant positions of the mantissa of a floating-point number, the results could still be inside the intrinsic variance of floating-point operations. Moreover, some physical simulations accept them as correct values in a given range, which can be as high as 4% for wave simulations [14]. Additionally, imprecise computation is gaining interest in various HPC applications [7]. In Section 4.4 we show that, depending on the application, the FIT rate could be significantly reduced if a small variance in the expected output is tolerated.

2.2 Related Work

Particle accelerators have been used for many years to measure and study radiation sensitivities for components and applications. Software fault injection (SFI) has also been widely used to estimate and study the resilience of hardware and applications. RTL level studies will not be considered as RTL level descriptions of HPC devices are not publicly available.

Cher et al. [12] use both proton irradiation and SFI to study the soft error resilience of BlueGene/Q. Proton bombardment shows that BG/Q has a mean time between correctable errors of 1.5 days validating the need for detection mechanisms. The SFI study was performed to analyze the behavior of applications without the mitigation mechanism employed on BG/Q hardware. Single and multiple bitflips are the two fault models used in the SFI study.

Oliveira et al. [38] study the radiation effect on NVIDIA and Intel accelerators using beam tests. The study defines metrics to quantify and qualify the radiation effects observing the magnitude and how the error spread in the final output. They state that simple mismatch detection cannot accurately evaluate the radiation sensitivity of

modern devices and algorithms. In this paper we take advantage of some metrics proposed in [38] to qualify SDCs, extending them to better consider output errors tolerance in HPC.

Binary Instrumentation-based Fault Injection Tool (BIFIT) is based on PIN and was proposed by Li et al. [32]. BIFIT was implemented to gather more information about when and where the fault is injected. BIFIT uses only the single bitflip model and injects faults based on memory region (Global, Heap, and Stack). The study finds that the time and location of faults are tightly related to applications output. For instance, global objects present a higher sensitivity and should be well protected.

GPU-Qin [19] is a fault injection tool based on the debugger mode in NVIDIA GPUs. Fang et al. propose GPU-Qin due the unacceptable simulation time of parallel devices like GPUs. The fault model used is only the single bitflip, and the study finds that algorithmic characteristics can help understand the variation in the SDC rates of applications.

SASSIFI [24] is a fault injector designed by NVIDIA that enables fault injection at a micro architectural level on GPUs. SASSIFI uses a low-level assembly-language tool called SASSI which inserts fault injection functions into the programs. Siva et al. argue that single and double bitflips alone cannot model the propagation of low-level faults to application level. Thus, random number and zero bits are included in the fault models.

Li et al. [33] extend the LLFI tool to profile and inject faults into GPU applications. They assume that memory and control logic are unlikely to experience errors, injecting single bitflips only in pipeline stages, flip-flops, ALUs, and register files. They observe that most faults do not propagate to other kernels and error propagation is application dependent.

The error propagation is the focus of Ashraf et al. in [3]. They propose a fault propagation framework using LLFI that keeps track of error propagation in MPI applications. Single bitflip is the fault model used to inject faults into the registers. They observe that a fault propagates linearly through time, and contaminates a consistent part of the output.

Unlike the described fault injectors, CAROL-FI injects errors at high-level to provide fast and pragmatic ideas to improve the reliability of HPC reliability by mitigating transient errors. We connect the reliability issues to the high-level source code which can help developers design resilient code and the deployment dedicated mitigation techniques.

3 METHODOLOGY

3.1 Tested Device

The Xeon Phi used is the coprocessor 3120A, as known as Knights Corner [27, 28]. The 3120A coprocessor has 57 physical in-order cores, and each one has 32 512-wide vector registers and supports four hardware threads. The device has a total of 6 GB GDDR5 main memory, and each core has 64 KB of L1 cache and 512 KB of L2 cache. The 3120A is built in a 22nm technology with Intel's 3-D Trigate transistors. The operating system is the CentOS 7.0 with Intel MPSS version 3.7 and GDB 7.8 with Intel extensions. The tested device is protected with Machine Check Architecture (MCA) reliability solution, which includes SECDED ECC in memory structures [28].

3.2 Selected Algorithms

To provide the experimental and analytical study of Xeon Phi, we selected six applications from different domains with different computation and communication patterns, following a general guideline for reliability studies [2, 39]. The selected benchmarks are: a DOE mini-app named ***CLAMR*** [22], a **Matrix Multiplication (*DGEMM*)** benchmark, and ***HotSpot***, ***LavaMD***, ***LUD*** and ***Needleman-Wunsch* (*NW*)** which are mini-apps from the Rodinia benchmark suite [10]. *NW* was only tested with our fault injection.

CLAMR is a DOE mini-application in the fluid dynamics domain and is representative of a LANL supercomputer workload. *CLAMR* simulates wave propagation using adaptive mesh refinement [22].

DGEMM is an optimized version of a matrix multiplication algorithm [2]. *DGEMM* is a compute-bound program that is often used to rank supercomputers.

HotSpot simulates the heat dissipation in an architectural floor plan to estimate processor temperature [10]. *HotSpot* is a memory-bound algorithm as its arithmetic intensity is low.

LavaMD implements an N-Body algorithm [2]. The algorithm analyzes particles in a 3D space and calculates the mutual forces between the particles within a predefined distance range [10].

LUD is a dense linear algebra like *DGEMM*. However, *LUD* uses less memory than *DGEMM* and has more interdependencies resulting in an algorithm that is less compute-bound than *DGEMM*.

Needleman-Wunsch (*NW*) is a dynamic programming algorithm developed to compare biological sequences [10]. It is representative of dynamic programming techniques that construct a new output using previous results.

4 BEAM EXPERIMENTS

This section presents the results of neutron beam experiments performed on the Xeon Phi. The main advantage of radiation experiments is that they provide a realistic evaluation of the processor error rate. By injecting faults in all the processor resources, beam experiments enable us to assess the actual FIT for different HPC benchmarks, which typically cannot be achieved by software fault injection only. Additionally, beam experiments can be used to evaluate how radiation errors propagate and affect the benchmark output. Still, software fault injection is used in Section 5 because during radiation experiments faults are observed only at the code output, limiting the observability and insights that can be directly applied for hardening solutions.

4.1 Neutron Beam Setup

Experiments were performed at the LANSCE facility, Los Alamos, NM, in October and December 2016. LANSCE flux is suitable to mimic the terrestrial neutron effects on electronic devices [48]. This means that error rates measured at LANSCE scaled down to the natural flux provide the predicted error rates on a realistic application expressed in Failure In Time (FIT). The experimental flux is about between $1 \times 10^5 n/(cm^2/s)$ and $2.5 \times 10^6 n/(cm^2/s)$, about 6 to 8 orders of magnitude higher than the atmospheric neutron flux at sea level (which is $13n/(cm^2/h)$ [29]). Tests were conducted for more than 500 hours of beam time. When scaled to the natural environment, our results cover at least 5×10^8 hours of normal operations, which are 57,000 years.

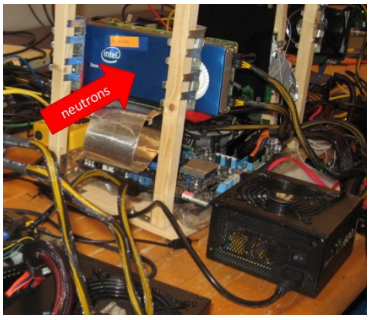


Figure 1: Part of the experimental setup.

In the real environment, having more than one fault at a time is rare. To maintain this behavior, experiments were tuned to guarantee observed output error rates lower than 10^{-4} errors/execution, ensuring that the probability of more than one neutron generating a failure in a single benchmark execution remains negligible.

On board DRAM data was not irradiated, allowing an analysis focused on the devices' core reliability. DRAM sensitivity, extensively studied [5, 11, 25, 30, 44], is out of the scope of this paper.

Figure 1 shows part of the experimental setup mounted at LANSCE. A host computer initializes the test sending pre-selected inputs to the accelerator and gathers results, comparing them with a pre-computed golden outputs. When a mismatch is detected, the execution is marked as affected by an error.

4.2 HPC Benchmarks Sensitivity

Figure 2 shows the results of the neutron beam experiments as the Failure in Time (FIT) rate for the five benchmarks executed on Xeon Phi at sea level (neutron flux increases with altitude). For each benchmark, we show the SDC FIT partitioned into up to the five observed output error patterns (discussed in the next subsection) and the DUE FIT. The SDC FIT includes all executions with any bit mismatch between the output of the program and the expected, error-free output. We collected more than 100 SDC/DUE for each benchmark, to have Normal's 95% confidence intervals lower than 10% of the presented values.

These results show large FIT variations between benchmarks and between SDCs and DUEs. For instance, *CLAMR* and *HotSpot* have their SDC FIT similar to their DUE FIT, while the other benchmarks show much smaller DUE FIT in comparison with their SDC FIT. This provides an initial insight that, for each DUE experienced for these benchmarks on a Xeon Phi, from one up to five SDCs will be observed.

If we extrapolate the FIT rates to a Trinity-size machine with 19,000 Xeon phi, operating at sea level, one should expect to see a SDC for *LUD* or DUE for *HotSpot* every eleven or twelve days. A hypothetical exascale machine built with the tested Xeon Phi would require at least an increase of 10× in the number of boards and would lead to almost daily SDC or DUE.

Benchmarks with several parallel iterations — such as *HotSpot* with its stencil convolution and *LUD* with its linear equations computation — working with single-precision floating-point matrices, exhibit the highest SDC FIT rates among the tested benchmarks. The other benchmarks show SDC FIT rates 30% to 75% smaller than

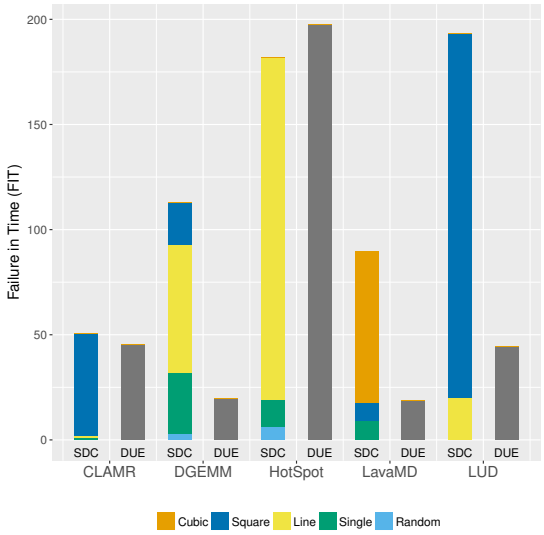


Figure 2: Benchmarks FIT and spatial distribution.

LUD. This can be due to a higher sensitivity to radiation in the data format, in the single-precision computation in the FPUs, or to a reduced capacity of these benchmarks to mask errors. Nevertheless, radiation experiments alone cannot provide the exact answer without additional (proprietary) details about the hardware.

HotSpot shows the highest DUE FIT rate among benchmarks. Its prevailing use of control flow statements and low arithmetic intensity seem to make it more prone to DUE. In contrast, more regular codes like *DGEMM* and *LavaMD* have the lowest DUE FITs.

4.3 Spatial Distribution of Errors

Our aim is to go beyond FIT rates calculations and extract more information from neutron beam experiments by investigating the spatial distribution of errors in the corrupted outputs. In each SDC FIT column in Figure 2, we categorize the outputs as having one of five failure patterns: (i) *single*, when a single output value is wrong; (ii) *line*, when more than one value in a row or column of an output matrix is wrong; (iii) *square*, when more than one value in two dimensions of an output matrix is wrong; (iv) *cubic*, when more than one value in three dimensions of the output matrices is wrong; and (v) *random*, when more than one value is wrong but with no clear pattern. Given that *LavaMD* is the only benchmark working with three dimensional simulations, it is the only one that can exhibit a cubic error pattern.

Results in Figure 2 indicate that most radiation-induced errors affect multiple values and dimensions of the outputs. It is worth noticing that, as described in Section 4.1, it is very unlikely to have more than one neutron generating fault in a single benchmark execution. Multiple output errors are then caused by a single particle corrupting multiple resources, by a corruption in a resource shared among parallel processes or corruptions that spread during computation, affecting multiple elements. The spread of errors is particularly significant in scientific iterative applications, where a value computed in one iteration is used as input for the next one [3, 33]. This fact maps well to benchmarks such as *CLAMR*,

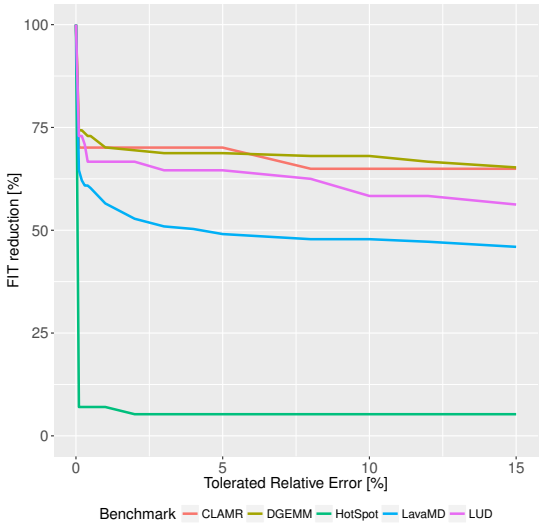


Figure 3: FIT decrease rate as a function of relative error tolerance.

LavaMD, and *LUD*. In other cases, intermediate values are computed and kept in local temporary memory before being used to compute output values, such as the case for *DGEMM*. Finally, multidimensional error patterns can also be a result of corruption affecting control flow instructions and internal variables, making a thread stop computing too soon or even wrongly overwriting the results of other threads.

Multi-dimensional errors patterns can be beneficial or detrimental for error detection and correction. For instance, the algorithm-based fault tolerance (ABFT) algorithm for matrix multiplication can correct single, line, and random errors in the output in O(1) time [26, 41]. Our results shows that for the Xeon Phi most of the observed SDCs in *DGEMM* could be corrected by ABFT. Meanwhile, a larger spread of errors could make it easier for some mechanisms to detect SDCs [6].

4.4 Relative Error

An additional important insight that can be gained from neutron beam experiments is how much different the values of the corrupted elements are from the expected, error-free, elements. The SDC FIT rates presented in Figure 2 consider as an error any bit mismatch between the experimental output and the expected output. However, there are some applications that can handle small output variations as natural imprecision from their methods. Additionally, floating-point operations have an intrinsic inaccuracy. Unfortunately, there is still no standard for how much an output can differ from its expected value. For example, Li et al. [33] consider any bit mismatch as an error with the exception of double-precision floating-point values, where comparisons are done using 40 digits of precision, while Ashrad et al. [3] use a 5% error margin in comparisons. Some simulations can tolerate up to 4% error margins, and imprecise computing is being applied to HPC [7, 14].

In Figure 3 we illustrate how varying the acceptable error margin affects the SDC FIT rates of the benchmarks. For each benchmark, we provide how much its SDC FIT rate changes (vertical axis) when we increase the acceptable error margin from 0.1% up to 15% (horizontal axis). Even a small acceptable error margin already decreases the SDC FIT rate of all benchmarks to at least 75% of its original value. For instance, this 25% drop makes the SDC FIT rate of *DGEMM* go from 113 to 84 (see Figure 2), which results in an increase of the MTBF (which is inversely proportional to FIT) by almost 35%. After this initial drop in FIT, decreases in the FIT rate saturate resulting in smaller improvements in the SDC FIT rates. This saturation is related to the way floating-point values are represented. For double-precision, a 0.1% error margin allows variations in 41 bits of the mantissa, while a 15% one allows variations in 49 bits of the mantissa.

The benchmark with the smallest relative error is *HotSpot*. Due to its stencil convolution behavior, errors usually spread to a significant portion of the output (resulting in a higher SDC FIT rate) but are also significantly attenuated. Even a small tolerated relative error of 2% results in a SDC FIT decrease to 5% of its original value, which means a twenty-fold increase in its SDC MTBF. This makes *HotSpot* the least sensitive benchmark to SDC.

CLAMR, which exhibited the lowest SDC FIT rate among the tested benchmarks, experiences (together with *DGEMM*) one of the smallest decreases in FIT rate with a relaxation of the error margin. This, coupled with the square error distribution, makes *CLAMR* most sensitive to radiation-induced errors. Still, the large output variations should make it easier for SDCs to be detected.

One final insight comes from combining the relative error and spatial distribution of errors analyses. As we have many instances of outputs with multiple wrong values and relative errors of over 15%, it seems that errors not only tend to propagate, but also tend to compound for four of the five benchmarks. Since these types of benchmarks do not provide natural means of error attenuation, detecting SDCs on Xeon Phi becomes vital to preventing large output errors and the spread of errors to many compute nodes.

5 FAULT INJECTION

We developed a high level fault injector to better understand transient errors propagation and provide useful insights to the code designer on how to mitigate their effects. Unlike the other available tools, we do not try to inject faults at the lowest possible level, but at the highest. Our goal, in fact, is not to measure the sensitivity to transient fault of an application, as we gathered this information with the neutron beam experiments in Section 4, but to identify the portions of the high-level code which are more critical for the application execution. We believe this information to be extremely useful for code developers.

5.1 CAROL-FI

We implemented a fault injector called CAROL-FI (available at [9]) to allow the injection of various fault models and correlate the injected faults with the algorithm structure. CAROL-FI is built upon GNU GDB. Debug information is used to correlate each allocated memory portion with its corresponding variable in the source code. Only compiling the code in debug mode allows to gather this information. As we are injecting at source code level, the fact that GDB impedes compiler optimizations does not undermine our results.

It is worth noting that GDB can also be used to inject faults in release mode, changing registers value and instruction bits. However, the goal of our study is to correlate the faults injected (and their outcomes) to particular portions of the source code, so we limit the use of CAROL-FI to debug mode and memory content corruption. In other words, the injection sites accessed by CAROL-FI consist of any source code variable allocated to a memory position.

Our fault injector is built upon two scripts. The first one, named **Supervisor**, is responsible for initiating GDB with the defined configurations (e.g., input parameters and the binary code). The Supervisor also works as a watchdog to kill the program if a user-defined time limit is surpassed. Finally, upon program execution completion, the Supervisor runs a user-defined function to check the output generated and log test data. The second script, named **Flip-script**, is called by GDB when the tested program is interrupted. Flip-script injects the fault into the program currently executing.

CAROL-FI's workflow is described as follows. The supervisor will initiate GDB, which will launch the code performing the fisrt step. Next, the Supervisor script will send the interrupt signal through the *killall* command after a random time. After the interrupt signal is captured by the program, GDB initiates the next step running the Flip-script. The Flip-script first selects one of the available threads and frames (which is the GDB's terminology for the call stack containing information of active process subroutines). Flip-script looks up the current frame upward the external one containing the global variables. Then, one of the variables of the selected frame will have its bits flipped. Such variables include *pointers*, *arrays*, *enums*, and *Integers*. After the memory address and offset of the selected data are known, Flip-script applies one of the fault models presented in Section 5.2. Then, Supervisor performs the final step, which kills the program if needed, and stores all the test data.

Finally, CAROL-FI logs the source code position that corresponds to the current instruction, the backtrace from GDB, the variable name, file name and line number where the variable is defined, the fault type applied, and the time window when the fault was injected.

CAROl-FI is very fast. On the average, its overhead is about 4× the normal execution time, with a worst case of 8×, as its only significant overheads are the ones caused by the GDB and the debug mode that disables compiler optimizations. There is no profiling phase and no breakpoints by GDB, in contrast to approaches like GPU-Qin, which can significantly increase the execution time. CAROL-FI executes the code at full speed until the interrupt signal is sent (GDB will not interact with the code). Once the program's execution stops, the GDB executes the flip functions with an execution time that varies according to how many subroutines are active and how many variables are allocated. Finally, the evaluated program will resume execution at full speed without further interaction from the GDB.

5.2 Fault Models

Our fault injection study does not distinguish between logic and memory errors. As the fault is generated at high level, by modifying the value of allocated memory, we are considering all possible transient faults that, by propagating from the transistor level, change the value of a memory location. These transient faults include errors that originated in memory, registers, caches, flip-flops in internal queues, control logic, etc. It is worth noting that identifying the individual probabilities of failures in the different logic and memory units is not feasible for components whose architecture details are not available. We use four different fault models to simulate the propagation of faults from low level to code level. The four models are:

- **Single**: flip a single random bit
- **Double**: flip two random bits
- **Random**: overwrite every bit by a random bit
- **Zero**: set every bit to zero

Single is the most commonly used fault model in the available fault-injectors, as described in Section 2.2. The double fault model is also often used when evaluating memory faults, as the probability of a single particle corrupting more than one word bit is not negligible [20]. SECDED ECC normally triggers application crash when a double bit error is detected. Our implementation of the Double model chooses two random bits located at the same byte offset, restricting the distance between the flipped bits. We emphasize that our Single and Double fault injections are not to be considered as faults in the memory alone, as those would be detected by ECC. We are simulating faults in all the unprotected resources that manifest in several ways at the highest level of abstraction. As shown in Section 4.2, the Xeon Phi error rate can be as high as 193 FIT, even if ECC is enabled. The probability of experiencing a corruption in unprotected resources that manifests at a high level is clearly not negligible.

Single and Double models are not considered sufficient for our purposes. Single bit faults are representative and adequate only if injected at the lowest accessible level and track how the original fault propagates to the microarchitecture level. Injections at a higher level require a more wide set of fault types to account for all possible effects of the original fault propagation. Thus, Single, Double, Random, and Zero models are a more representative set of the possible outcomes that a fault can manifest at a higher level.

6 FAULT INJECTION ANALYSIS

We have injected at least 10,000 faults into each of the selected benchmarks, which are sufficient to guarantee the worst case statistical error bars at 95% confidence level to be at most 1.96%. For each fault injection experiment, we collected the output of the program execution and compared it to a previously computed golden copy.

Figure 4 presents the percentage of faults that are masked or cause an SDC or DUE for each benchmark presented in Section 3.2. For most of the benchmarks, SDCs are less likely to occur than DUEs, while the majority of injected faults are masked during computation (except for *DGEMM*). As explained in Section 5, it is not possible to directly correlate our beam experiments with CAROL-FI results. Figure 4 shows the probability of corrupted portions of the source code to affect the execution, while Figure 2 shows the likelihood of neutrons to induce errors in physical transistors combined with the probability of those errors to propagate to the output.

Figures 5a and 5b show the Program Vulnerability Factor (PVF) for SDC and DUE, respectively, for each fault model described in Section 5.2. The different fault models yield quite different PVFs

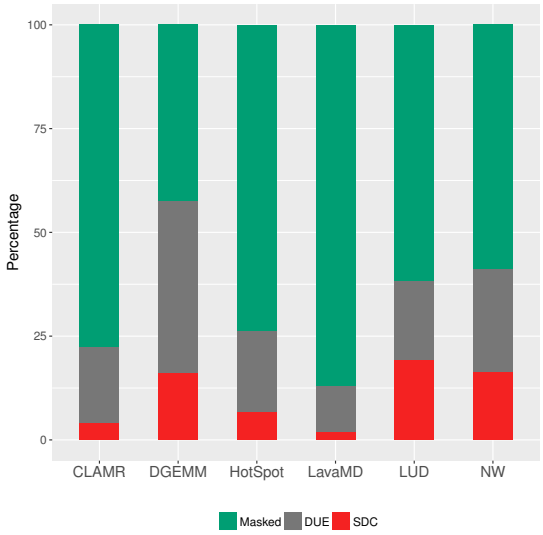
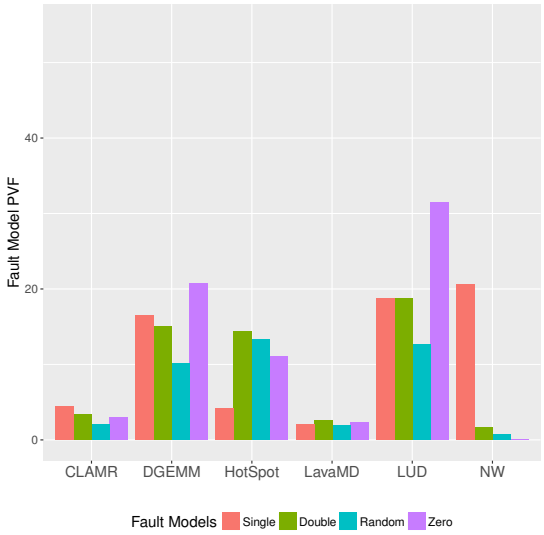
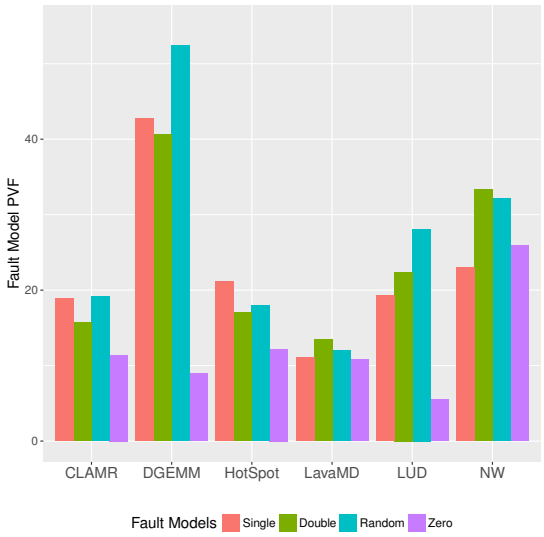


Figure 4: Outcomes of fault injections.



(a) SDC.



(b) DUE.

Figure 5: The PVF of the benchmarks for the different fault models.

depending on the benchmark application class and characteristics. For example, algebraic benchmarks like *DGEMM* and *LUD* have similar PVFs. The different models also affect the type of errors observed, for instance, the Zero model provides lower DUE.

To evaluate the dependence of the impact of faults on the timing of their occurrence, we divided the benchmarks into equal parts based on the execution time. The length of each part is selected to be short enough to provide insight into the injection time *vs* fault sensitivity, and long enough to allow a statistically significant amount of injections. *CLAMR* is divided into nine time windows of equal length. *DGEMM* and *HotSpot* are split into five time windows while *LUD* and *NW* are divided into four parts each. We then calculated the percentage of faults injected into each time window that caused an SDC or DUE (shown in Figures 6a and 6b, respectively). Please note that Figures 6a and 6b show the PVF for each time window, not to be confused with the contribution of each time window to the benchmark PVF, which is why the sum of percentages is higher than 100%.

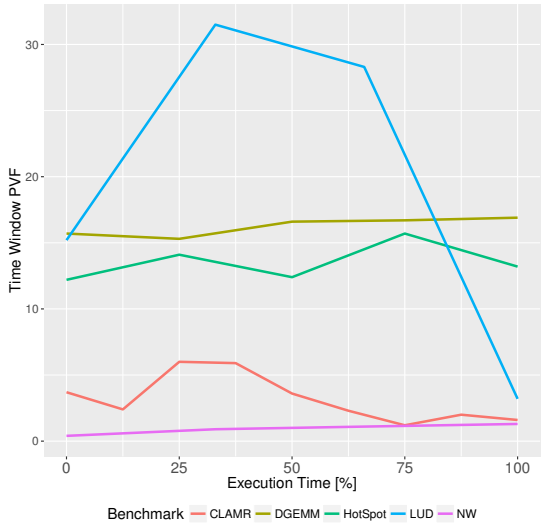
In the following we analyze each benchmark individually to demonstrate how CAROL-FI can be used to derive information about benchmark sensitivity and also provide insights on how to improve the resilience of each benchmark.

DGEMM

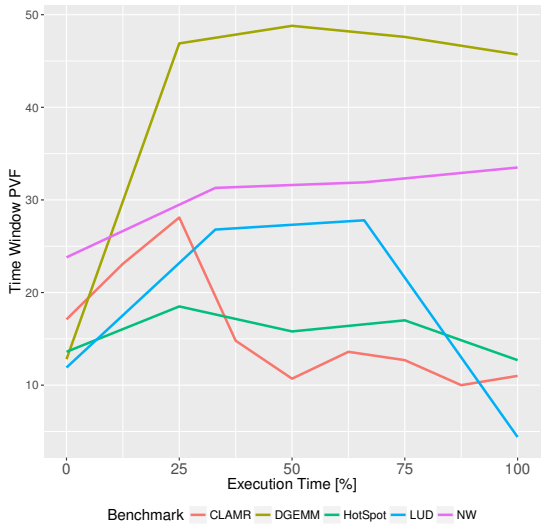
As shown in Figure 4, about 60% of the faults injected in *DGEMM* generate an error (SDC or DUE). Most of the observed SDCs and DUEs are the result of faults injected into the input and output **matrices** and **control variables**.

Faults injected in the matrices caused SDCs and DUEs 43% and 19% of the times, respectively. For control variables, 38% of the faults injected generate SDCs and 38% cause DUEs. *DGEMM* creates nine loop control variables of integer type, which may seem to be a negligible number and, thus, unlikely to be corrupted. 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, increasing the memory portion used to store them. In contrast, the memory portion used to store the matrices remains the same regardless of the level of parallelism. As a result, the probability of having a corrupted loop control variable becomes significant, and the severity of that corruption is very high.

Evaluating the fault models, we observe in Figures 5a and 5b that the Single and Double models have a similar outcome. On the other hand, the Random model exhibits a lower SDC error rate while the Zero model has a higher one. Observing the DUE rate in Figure 5b, we find that Random and Zero have opposite behaviors. Random and Zero models have a higher likelihood to



(a) SDC.



(b) DUE.

Figure 6: The dependence of the PVF of the benchmarks on the execution time window.

generate largely different values than the expected ones. However, we believe that the Random converts some SDCs to DUEs since the corrupted values can be used to access invalid memory addresses, invalid indexes, or another operation that will lead to a DUE. The Zero model, in contrast, generates values that will most likely cause an SDC instead of a DUE.

The *DGEMM* benchmark has the same memory and resources usage during the entire execution. Therefore, the time window dependent sensitivity in Figure 6a shows that the SDC error rate remains unchanged between time windows. However, in Figure 6b we see that *DGEMM* DUE rate is lower at the beginning when the program is still initializing and control flow operations are less common.

Protecting the control variables can lead to a significant impact in the final DUE rate. Selective duplication with comparison can be applied to protect the internal memory structures that contain such control variables. ECC or parity implemented to protect all memory strutctures will detect or even correct such errors but, to improve the resilience at a lower overhead, a selective protection should be preferred.

Additionally, logic errors that modify the result of instructions that update loop control variables are likely to impact the output and could not be detected with ECC but could be detected by residue module check.

CLAMR

Injected faults are masked 75% of the time in *CLAMR*, as shown in Figure 4. CAROL-FI identifies **mesh** to be the most critical portion of the benchmark. We can divide the mesh operations into three parts: **Sort**, **Tree**, and **others**.

Of all the injections in *Sort*, 39% generate an SDC and 43% cause DUEs. The *Tree* part of *CLAMR* includes the functions responsible for the creation and operation of a K-D Tree. 20% of all the faults in *Tree* generate an SDC and 41% cause a DUE. All the faults in the remaining variables of the mesh code are classified as *others*. Only 33% of the faults in this part generate an SDC and 28% cause DUEs.

The fault models show similar rates for *CLAMR* SDCs (see Figure 5a). For DUEs, only the Zero model yields a different rate than the other models, as can be seen in Figure 5b. The reason for this is the same for *DGEMM*, where zero values are less likely to generate errors that cause a DUE.

We can observe in Figure 6 that time window 3 exhibits the highest error rate and then it decreases. This behavior is similar to the one observed when using a more low-level fault injection in [22]. *CLAMR* becomes more sensitive when the number of active cells reaches its maximum value, which can be automatically set by the algorithm itself.

Our fault injection analysis shows that *Mesh* operations and structure are the most sensitive portions of *CLAMR*, which is expected since it is the main structure used to define and hold the system data. Furthermore, *Sort* and *Tree* operations are equally sensitive to DUEs, causing the majority of the harmful outcomes. However, for SDCs, *Sort* has double the sensitivity and should have a higher priority when attempting to improve reliability. Thus, specific techniques targeting Sort [1] and Tree operations can improve the overall resilience of *CLAMR*. Additionally, general techniques like redundant multithreading applied only to those critical functions and operations may also yield an improved resilience with a fair overhead. Moreover, by reducing the DUE rate caused by fault in Sort and Tree, HPC systems can allow lowering the frequency of checkpointing techniques.

HotSpot

HotSpot shows trends similar to *CLAMR*. About 75% of the faults are masked and do not affect the output, as shown in Figure 4. Most of the observed SDCs and DUEs are caused by injections in **constant and control variables** used during computation. Our fault injection analysis shows that about 30% of the faults in control and constant variables cause an SDC and 40% generate a DUE.

HotSpot is a stencil algorithm like *CLAMR*, but *HotSpot* simulates an open system. Thus, SDCs in the program can be dissipated out of the system given enough iterations. Out of the four fault models, the Single model has the highest chance to introduce small errors since it flips only one bit, while the other fault models flip two or more bits. We can see in Figure 5a that the Single model has indeed the lowest error rate, showing the *HotSpot* ability to recover from it. Considering DUEs, the Single model has the same outcome as the Double and Random ones. The Zero model has the lowest rate since any bit flipped using the other models can lead to invalid operations while Zero will likely cause an SDC.

Similarly to *DGEMM*, *HotSpot* keeps the memory and resources utilization around the same level during the execution. Therefore, the sensitivity for each time window deviates only by a small amount as can be seen in Figures 6a and 6b.

HotSpot computes the temperature of functional blocks in a chip when executing a program, given the power consumed by these blocks. The temperatures of the different blocks are calculated in an iterative manner and thus, errors in intermediate values will have negligible effect on the final results. This computing strategy is intrinsically robust to data errors. In fact, the impact of a fault on the value of a variable will be reduced by the use of nearby correct values in subsequent iterations. Taking advantage of the intrinsic robustness of the algorithm, we can focus hardening efforts on the variables that the fault injection campaign has shown to be more sensitive. Thus, applying a simple replication of the sensitive variables will yield a better performance/reliability ratio than a more comprehensive strategy.

LavaMD

Figure 4 shows that, for *LavaMD*, only 15% of the injected faults produce an SDC or DUE. Faults in the **charge and distance arrays** and **control variables** cause the vast majority of harmful outcomes.

The *charge* and *distance* arrays used in the algorithm are responsible for 57% of the SDCs and 11% of the DUEs. The two arrays are up to five orders of magnitude larger than the other data structures that cause harmful effects. Thus, the probability of a fault to occur in the two input arrays is higher than for the other data structures. Therefore, these two arrays are the most critical parts of the benchmark.

Figures 5a and 5b show that the four fault models have similar results for SDCs and DUEs in *LavaMD*. *LavaMD* is a complex algorithm, and the impact of each fault depends on several factors such as the item (particle) corrupted, position in the 3D space, and the state of neighboring particles. However, the fault model and magnitude of the corrupted element seem to have the same impact due the nature of the operations performed. *LavaMD* executes exponentiation operation, and this will exacerbate any error.

LavaMD presents one of the biggest challenges to devise a hardening technique that can significantly improve resilience without compromising performance. In fact, a large amount of memory is exposed to corruption that is likely to generate an SDC or DUE. Thus, unless a specific technique for *LavaMD* is developed, a generic technique, like modular replication and checkpointing should be applied, which may consume up to twice the execution time and energy.

LUD

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 rates for *LUD* are well-balanced, and *LUD* has a much lower DUE rate than *DGEMM* (see Figure 4).

Faults in the main matrix and the temporary matrices allocated during the computation of the decomposition generate an SDC for about 54% of the injected faults and 28% cause a DUE. Evaluating the control variables, we observe that 24% of the faults generate an SDC and 36% cause a DUE.

Figures 5a and 5b show that the fault models have a similar behavior for algebraic algorithms like *LUD* and *DGEMM*. The Random and Zero models seem to shift some SDCs to DUEs and vice versa. This similarity indicates that the fault models behavior among algorithms from the same class can be similar, and the same insights obtained from one benchmark can be applied to a larger number of algorithms.

LUD has many row and column interdependencies resulting in a higher load in the middle of execution, which also corresponds to the more critical time windows. Therefore, while the fault model behavior is dependent on the algorithm class, the time window sensitivity is associated with the workload computed during that time.

To mitigate errors in *LUD* we can take advantage of the time dependent sensitivity and use a heavier mitigation technique in the middle of the execution and a lighter one in the beginning and end. Moreover, we can rely on residue check for matrix operations and apply redundant multithreading or duplication with comparison to control variables, improving reliability without compromising performance too much.

NW

NW has a well-balanced rate between SDC and DUE, as Figure 4 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 SDC and DUE have a similar probability to occur when a fault is injected in the matrices.

NW is the only algorithm using integers as its main data type. We can see in Figure 5a that the Zero faults do not cause any errors. *NW* dynamically constructs a matrix based on matches and mismatches of the input values, and a large portion of the matrix and values manipulated will be zero. Thus, the Zero faults have the highest chance to be masked. Double and Random have the highest probability to introduce significantly different values in *NW* since the algorithm works with small and zero values. Still, Single is the fault model with the highest rate of SDCs in *NW* while Double and Random result in very few SDCs, but when we look into DUE (refer to Figure 5b), Double and Random have the highest error rate for *NW*. Thus, *NW* will most likely crash when the value is largely different from the expected one.

NW presents a lower DUE rate in the beginning when the algorithm has a limited workload to compute. After the workload reaches its highest value, the sensitivity at each time window stabilizes for DUEs and SDCs.

Similarly to *LavaMD*, *NW* presents a considerable challenge to hardening if one wishes to protect all the sensitive memory

which is most of the memory used by the algorithm. The source of SDCs and DUEs is the same, i.e., faults in the matrices. Thus, protecting the matrices will improve both rates. Residue check and control flow techniques may provide a good reliability without a high degradation in performance.

6.1 Discussion

As we can see from radiation data in Figure 2, the actual FIT rate is already too high even with ECC in most memory structures. Internal queues, flip-flops, or even logic circuits, are not protected, and errors in these parts will propagate to memory. Furthermore, errors in these unprotected parts, especially the logic circuit, can manifest in different ways such as random or zero values. The overhead to protect from all the fault types can be too costly. Thus, we can evaluate the most critical code portions, fault models, and time windows for each class of application and apply the most appropriate level of protection to provide the desired level of resilience.

Algebraic applications can be better protected with residue error detection than ECC, which is unable to correct Random or Zero faults nor the logic circuit. We need only 8 bits to use $mod15$ for the residue error protection, or only 2 bits for $mod3$. Residue protection can also be applied to hardware providing fast mechanisms using small portions of chip area.

For *NW*, a simple parity would detect most SDCs since single faults are more critical than the others types of faults. Therefore, the ability to disable or to provide weaker mitigation mechanisms will significantly improve the performance and sustain the desired level of resilience.

For applications like *HotSpot* and *CLAMR*, we can take into consideration the natural resilience of the algorithm, especially when allowing a certain percentage of tolerated error (see Figure 3) so a simple mitigation technique can provide the desired level of resilience.

7 CONCLUSION AND FUTURE WORK

We present the realistic SDC and DUE rate of Xeon Phis. We go beyond the sole FIT rate and also evaluate how the errors spread and the severity of SDCs. We demonstrate that output error patterns can be beneficial to evaluate the efficacy of mitigation techniques like ABFT, which can detect and correct errors depending on the spatial locality of the errors. We also investigate how the notion of imprecise computation can be applied to HPC applications and measure the FIT rate decrease as a function of accepted error tolerance.

We show fault injection analysis to correlate SDCs and DUEs with the high-level code, improving the understanding of applications reliability. CAROL-FI identifies which portions of the code are more prone to be corrupted and cause an SDC or DUE. We also observe that, for some programs, the probability of corruption to propagate significantly depends on the time window in which the fault occurs. Additionally. we have studied the severity of various fault type (i.e. Single, Double, Random, or Zeros). We believe that the reported radiation and fault injection analysis provide pragmatic information to design the best hardening solution for scientific applications, balancing the resilience and performance.

In the future, we plan to implement the mitigation techniques based on the radiation and fault injection analysis. Then, we will validate them with radiation experiments and fault injection campaigns.

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. This work was also supported by the CNPq project grant 454698/2014-3 and STIC-AmSud/CAPES scientific cooperation program under the EnergySFE research project grant 99999.007556/2015-02. A portion of this work was performed at the Ultrascale Systems Research Center (USRC) at Los Alamos National Laboratory, supported by the U.S. Department of Energy contract AC52-06NA25396. The publication has been assigned the LANL identifier LA-UR-17-23185.

REFERENCES

[1] C. A. Argyrides, C. A. Lisboa, D. K. Pradhan, and L. Carro. 2009. Single element correction in sorting algorithms with minimum delay overhead. In *2009 10th Latin American Test Workshop*. 1–6. https://doi.org/10.1109/LATW.2009.4813812

[2] Krste Asanovic et al. 2006. *The Landscape of Parallel Computing Research: A View from Berkeley*. Technical Report UCB/EECS-2006-183. EECS Department, University of California, Berkeley. http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html

[3] Rizwan A. Ashraf, Roberto Gioiosa, Gokcen Kestor, Ronald F. DeMara, Chen-Yong Cher, and Pradip Bose. 2015. Understanding the Propagation of Transient Errors in HPC Applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*. ACM, New York, NY, USA, Article 72, 12 pages. https://doi.org/10.1145/2807591.2807670

[4] R.C. Baumann. 2005. Radiation-induced soft errors in advanced semiconductor technologies. *Device and Materials Reliability, IEEE Transactions on* 5, 3 (Sept 2005), 305–316. https://doi.org/10.1109/TDMR.2005.853449

[5] R. Baumann. 2005. Soft errors in advanced computer systems. *Design Test of Computers, IEEE* 22, 3 (2005). https://doi.org/10.1109/MDT.2005.69

[6] Eduardo Berrocal, Leonardo Bautista-Gomez, Sheng Di, Zhiling Lan, and Franck Cappello. 2016. *Exploring Partial Replication to Improve Lightweight Silent Data Corruption Detection for HPC Applications*. Springer International Publishing, Cham, 419–430. https://doi.org/10.1007/978-3-319-43659-3_31

[7] Melvin A Breuer. 2005. Multi-media applications and imprecise computation. In *Digital System Design, 2005. Proceedings. 8th Euromicro Conference on*. IEEE, 2–7.

[8] S. Buchner, M. Baze, D. Brown, D. McMorrow, and J. Melinger. 1997. Comparison of error rates in combinational and sequential logic. *Nuclear Science, IEEE Transactions on* 44, 6 (1997), 2209–2216.

[9] Carol-FI Fault Injector 2017. Carol-FI. https://github.com/UFRGS-CAROL/carol-fi. (2017).

[10] Shuai Che, M. Boyer, Jiayuan Meng, D. Tarjan, J. W. Sheaffer, Sang-Ha Lee, and K. Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. 44–54. http://dx.doi.org/10.1109/IISWC.2009.5306797

[11] H. M. Chen, S. Jeloka, A. Arunkumar, D. Blaauw, C. J. Wu, T. Mudge, and C. Chakrabarti. 2016. Using Low Cost Erasure and Error Correction Schemes to Improve Reliability of Commodity DRAM Systems. *IEEE Trans. Comput.* 65, 12 (Dec 2016), 3766–3779. https://doi.org/10.1109/TC.2016.2550455

[12] Chen-Yong Cher, Meeta S. Gupta, Pradip Bose, and K. Paul Muller. 2014. Understanding Soft Error Resiliency of BlueGene/Q Compute Chip Through Hardware Proton Irradiation and Software Fault Injection. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14)*. IEEE Press, Piscataway, NJ, USA, 587–596. https://doi.org/10.1109/SC.2014.53

[13] C. Constantinescu. 2002. Impact of deep submicron technology on dependability of VLSI circuits. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*. 205–209. https://doi.org/10.1109/DSN.2002.1028901

[14] Josep de la Puente, Miguel Ferrer, Mauricio Hanzich, JosÃĪ E. Castillo, and JosÃĪ M. Cela. 2014. Mimetic seismic wave modeling including topography on deformed staggered grids. *GEOPHYSICS* 79, 3 (2014), T125–T141. https://doi.org/10.1190/geo2013-0371.1 arXiv:http://dx.doi.org/10.1190/geo2013-0371.1

[15] D. A. G. de Oliveira, L. L. Pilla, T. Santini, and P. Rech. 2016. Evaluation and Mitigation of Radiation-Induced Soft Errors in Graphics Processing Units. *IEEE Trans. Comput.* 65, 3 (March 2016), 791–804. https://doi.org/10.1109/TC.2015.2444855

[16] Nathan DeBardeleben, Sean Blanchard, Laura Monroe, Phil Romero, Daryl Grunau, Craig Idler, and Cornwell Wright. 2013. GPU Behavior on a Large HPC Cluster. *6th Workshop on Resiliency in High Performance Computing (Resilience)*

in Clusters, Clouds, and Grids in conjunction with the 19th International European Conference on Parallel and Distributed Computing (Euro-Par 2013), Aachen, Germany, (August 26-30 2013).

[17] J.J. Dongarra, H.W. Meuer, and E. Strohmaier. 2016. TOP500 Supercomputer Sites: June 2016. (2016). http://www.top500.org

[18] B. Fang, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi. 2014. 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*. 221–230. https://doi.org/10.1109/ISPASS.2014.6844486

[19] Bo Fang, Karthik Pattabiraman, Matei Ripeanu, and Sudhanva Gurumurthi. 2014. 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, 221–230.

[20] Y. P. Fang and A. S. Oates. 2016. Characterization of Single Bit and Multiple Cell Soft Error Events in Planar and FinFET SRAMs. *IEEE Transactions on Device and Materials Reliability* 16, 2 (June 2016), 132–137. https://doi.org/10.1109/TDMR.2016.2535663

[21] L. A. Bautista Gomez, Franck Cappello, L. Carro, N. DeBardeleben, B. Fang, S. Gurumurthi, S. Keckler, K. Pattabiraman, R. Rech, and M. Sonza Reorda. 2014. GPGPUs: How to Combine High Computational Power with High Reliability. In *2014 Design Automation and Test in Europe Conference and Exhibition*. Dresden, Germany.

[22] Qiang Guan, N. DeBardeleben, B. Artkinson, R. Robey, and W.M. Jones. 2015. 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*. 176–179. https://doi.org/10.1109/CLUSTER.2015.35

[23] Siva Kumar Sastry Hari et al. 2015. SASSIFI: Evaluating Resilience of GPU Applications. In *Proceedings of the Workshop on Silicon Errors in Logic-System Effects (SELSE)*.

[24] Siva Kumar Sastry Hari, Timothy Tsai, Mark Stephenson, Stephen W. Keckler, and Joel Emer. 2017. SASSIFI: An Architecture-level Fault Injection Tool for GPU Application Resilience Evaluation. *International Symposium on Performance Analysis of Systems and Software* (Oct 2017).

[25] S. M. Hassan, W. J. Song, S. Mukhopadhyay, and S. Yalamanchili. 2016. Reliability-performance tradeoffs between 2.5D and 3D-stacked DRAM processors. In *2016 IEEE International Reliability Physics Symposium (IRPS)*. MY–2–1–MY–2–5. https://doi.org/10.1109/IRPS.2016.7574618

[26] Kuang-Hua Huang and J.A. Abraham. 1984. Algorithm-Based Fault Tolerance for Matrix Operations. *Computers, IEEE Transactions on* C-33, 6 (June 1984), 518–528. https://doi.org/10.1109/TC.1984.1676475

[27] Intel. 2016. An Overview of Programming for Intel Xeon processors and Intel Xeon Phi coprocessors. (2016). http://download.intel.com/newsroom/kits/xeon/phi/pdfs/overview-programming-intel-xeon-intel-xeon-phi-coprocessors.pdf

[28] Intel. 2016. Intel Xeon Phi Coprocessor System Software Developers Guide. (2016). https://software.intel.com/sites/default/files/managed/09/07/xeon-phi-coprocessor-system-software-developers-guide.pdf

[29] JEDEC. 2006. *Measurement and Reporting of Alpha Particle and Terrestrial Cosmic Ray-Induced Soft Errors in Semiconductor Devices*. Technical Report JESD89A. JEDEC Standard.

[30] Saeng-Hwan Kim, Won-Oh Lee, Jung-Ho Kim, Seong-Seop Lee, Sun-Young Hwang, Chang-Il Kim, Tae-Woo Kwon, Bong-Seok Han, Sung-Kwon Cho, Dae-Hui Kim, Jae-Keun Hong, Min-Yung Lee, Sung-Wook Yin, Hyeon-Gon Kim, Jin-Hong Ahn, Yong-Tark Kim, Yo-Hwan Koh, and Joong-Sik Kih. 2007. A low power and highly reliable 400Mbps mobile DDR SDRAM with on-chip distributed ECC. In *Solid-State Circuits Conference, 2007. ASSCC '07. IEEE Asian*. 34–37. https://doi.org/10.1109/ASSCC.2007.4425789

[31] J. C. Laprie. 1995. DEPENDABLE COMPUTING AND FAULT TOLERANCE : CONCEPTS AND TERMINOLOGY. In *Fault-Tolerant Computing, 1995, Highlights from Twenty-Five Years., Twenty-Fifth International Symposium on*. 2–. https://doi.org/10.1109/FTCSH.1995.532603

[32] Dong Li, Jeffrey S. Vetter, and Weikuan Yu. 2012. 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 (SC '12)*. IEEE Computer Society Press, Los Alamitos, CA, USA, Article 57, 11 pages. http://dl.acm.org/citation.cfm?id=2388996.2389074

[33] Guanpeng Li, Karthik Pattabiraman, Chen-Yong Cher, and Pradip Bose. 2016. Understanding Error Propagation in GPGPU Applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '16)*. IEEE Press, Piscataway, NJ, USA, Article 21, 12 pages. http://dl.acm.org/citation.cfm?id=3014904.3014932

[34] Robert Lucas. 2014. Top Ten Exascale Research Challenges. In *DOE ASCAC Subcommittee Report*.

[35] R. R. Lutz. 1993. Analyzing software requirements errors in safety-critical, embedded systems. In *Requirements Engineering, 1993., Proceedings of IEEE International Symposium on*. 126–133. https://doi.org/10.1109/ISRE.1993.324825

[36] N.N. Mahatme, T.D. Jagannathan, L.W. Massengill, B.L. Bhuva, S.-J. Wen, and R. Wong. 2011. Comparison of Combinational and Sequential Error Rates for a Deep Submicron Process. *Nuclear Science, IEEE Transactions on* 58, 6 (2011), 2719–2725.

[37] M. Nicolaidis. 1999. Time redundancy based soft-error tolerance to rescue nanometer technologies. In *VLSI Test Symposium, 1999. Proceedings. 17th IEEE*. 86–94. https://doi.org/10.1109/VTEST.1999.766651

[38] D. Oliveira, L. Pilla, M. Hanzich, V. Fratin, F Fernandes, C. Lunardi, J. Cela, P. Navaux, L. Carro, and P. Rech. 2017. Radiation-Induced Error Criticality in Modern HPC Parallel Accelerators. In *Proceedings of 21st IEEE Symp. on High Performance Computer Architecture (HPCA)*. ACM.

[39] H. Quinn, W. H. Robinson, P. Rech, M. Aguirre, A. Barnard, M. Desogus, L. Entrena, M. Garcia-Valderas, S. M. Guertin, D. Kaeli, F. L. Kastensmidt, B. T. Kiddie, A. Sanchez-Clemente, M. S. Reorda, L. Sterpone, and M. Wirthlin. 2015. Using Benchmarks for Radiation Testing of Microprocessors and FPGAs. *IEEE Transactions on Nuclear Science* 62, 6 (Dec 2015), 2547–2554. https://doi.org/10.1109/TNS.2015.2498313

[40] Radiation Experiment Results 2017. Radiation Experiment Results. https://github.com/UFRGS-CAROL/sc17-log-data. (2017).

[41] P. Rech, C. Aguiar, C. Frost, and L. Carro. 2013. An Efficient and Experimentally Tuned Software-Based Hardening Strategy for Matrix Multiplication on GPUs. *Nuclear Science, IEEE Transactions on* 60, 4 (2013), 2797–2804. https://doi.org/10.1109/TNS.2013.2252625

[42] Advanced Scientific Computing Research. 2016. Scientific Discovery through Advanced Computing - The Challenges of Exascale. https://science.energy.gov/ascr/research/scidac/exascale-challenges/. (2016). [Online; accessed 5-March-2016].

[43] G. P. Saggese, N. J. Wang, Z. T. Kalbarczyk, S. J. Patel, and R. K. Iyer. 2005. An experimental study of soft errors in microprocessors. *IEEE Micro* 25, 6 (Nov 2005), 30–39. https://doi.org/10.1109/MM.2005.104

[44] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. 2009. DRAM Errors in the Wild: A Large-Scale Field Study. In *SIGMETRICS*.

[45] Marc Snir, Robert W Wisniewski, Jacob A Abraham, Sarita V Adve, Saurabh Bagchi, Pavan Balaji, Jim Belak, Pradip Bose, Franck Cappello, Bill Carlson, et al. 2014. Addressing failures in exascale computing. *International Journal of High Performance Computing Applications* (2014), 1–45.

[46] Devesh Tiwari, Saurabh Gupta, Jim Rogers, Don Maxwell, Paolo Rech, Sudharshan Vazhkudai, Daniel Oliveira, Dave Londo, Nathan Debardeleben, Philippe Navaux, Luigi Carro, and Arthur Buddy Bland. 2015. 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.

[47] S. Tselonis and D. Gizopoulos. 2016. GUFI: A framework for GPUs reliability assessment. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 90–100. https://doi.org/10.1109/ISPASS.2016.7482077

[48] M. Violante, L. Sterpone, A. Manuzzato, S. Gerardin, P. Rech, M. Bagatin, A. Paccagnella, C. Andreani, G. Gorini, A. Pietropaolo, G. Cardarilli, S. Pontarelli, and C. Frost. 2007. A New Hardware/Software Platform and a New 1/E Neutron Source for Soft Error Studies: Testing FPGAs at the ISIS Facility. *Nuclear Science, IEEE Transactions on* 54, 4 (2007), 1184–1189. https://doi.org/10.1109/TNS.2007.902349

[49] H.-J. Wunderlich, C. Braun, and S. Halder. 2013. Efficacy and efficiency of algorithm-based fault-tolerance on GPUs. In *On-Line Testing Symposium (IOLTS), 2013 IEEE 19th International*. 240–243. https://doi.org/10.1109/IOLTS.2013.6604090

A ARTIFACT DESCRIPTION: EXPERIMENTAL AND ANALYTICAL STUDY OF XEON PHI RELIABILITY

A.1 Abstract

This artifact description details how to obtain all the software used in this work as well as how to execute it to reproduce the fault injection campaigns presented in Section 6. It assumes that the OS and the Intel software is correctly installed.

A.2 Description

A.2.1 Check-list (artifact meta information).

- **Algorithm:** *CLAMR*, *DGEMM*, *HotSpot*, *LavaMD*, *LUD*, and *NW*
- **Program:** C, C++, and OpenMP code
- **Compilation:** ICC 16.0.2 20160204
- **Data set:** Dynamically generated or provided with the code
- **Run-time environment:** CentOS 7, Intel MPSS 3.7, and GNU GDB 7.8-16.0.677

- **Hardware:** Intel Xeon Phi 3120A
- **Experiment workflow:** Download the repository, compile the source code, execute the automated scripts, and run the parse scripts
- **Publicly available?:** Yes

A.2.2 How software can be obtained (if available). Carol-FI and benchmark code can be cloned from GitHub (available in [9]).

A.2.3 Hardware dependencies. Intel Xeon Phi 3120A or similar Knights Corner hardware. The underlying system has no restriction once the Intel compiler will produce native code to be executed on the Knights Corner device.

A.2.4 Software dependencies. Intel MPSS 3.7 must be installed as well as Intel Parallel Studio 2016 to provide the required tools to compile and run the software.

A.2.5 Datasets. HotSpot is the only code that uses predefined datasets. The other benchmarks generate dynamic datasets that will be generated once and used during the whole fault injection campaign.

A.3 Installation

There is no additional software to install besides the Intel software stack referenced in A.2.4. There are instructions and a Makefile for each algorithm's source code that was used in this work. There is also a README file that explains step by step how to run the fault injector in a generic machine and the Intel Xeon Phi.

A.4 Experiment workflow

The first step is to compile the source code with debug information. Then, a configuration file is produced with all the information needed by the fault injector. Finally, the fault injector is executed with the configuration file as an argument and how many times the experiment should be repeated.

A.5 Evaluation and expected result

All the results are located in [40]. The parser scripts are located in the *parser-scripts* folder, a README file is also provided to detail how to execute and how to interpret the results produced.

A.6 Experiment customization

A detailed README showing how to execute in different environments and with different benchmarks is also included in the repository.