UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

DANIEL ALFONSO GONÇALVES DE OLIVEIRA

Hardening Strategies for HPC Applications

Thesis presented in partial fulfillment
of the requirements for the degree of
Doctor of Computer Science

Advisor: Prof. Dr. Philippe Olivier Alexandre
Navaux
Coadvisor: Prof. Dr. Paolo Rech

Porto Alegre
May 2018

CIP — CATALOGING-IN-PUBLICATION

Oliveira, Daniel Alfonso Gonçalves de

Hardening Strategies for HPC Applications / Daniel Alfonso Gonçalves de Oliveira. – Porto Alegre: PPGC da UFRGS, 2018.

133 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2018. Advisor: Philippe Olivier Alexandre Navaux; Coadvisor: Paolo Rech.

1. HPC. 2. Fault Tolerance. 3. Accelerators. 4. Radiation Experiments. 5. Fault Injection. 6. Reliability. 7. Hardening Strategies. 8. Selective Hardening. I. Navaux, Philippe Olivier Alexandre. II. Rech, Paolo. III. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Rui Vicente Oppermann
Vice-Reitora: Prof[a]. Jane Fraga Tutikian
Pró-Reitor de Pós-Graduação: Prof. Celso Giannetti Loureiro Chaves
Diretora do Instituto de Informática: Prof[a]. Carla Maria Dal Sasso Freitas
Coordenador do PPGC: Prof. João Luiz Dihl Comba
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

ACKNOWLEDGEMENT

First of all, I would like to thank Prof. Philippe Olivier Alexandre Navaux, my Ph.D. advisor, and Prof. Paolo Rech, my Ph.D. coadvisor. Thanks for all the support and patience during my academic development. The precious advisement helped me reach farther than I could imagine.

I would also like to thank Prof. Israel Koren, who was my advisor on a Sandwich Ph.D. at the University of Massachusetts at Amherst. Thanks for receiving me and all the ideas and discussions regarding this thesis, especially the CAROL-FI fault injector that was developed during my time at UMass Amherst. I cannot forget Prof. Sandip Kundu who also helped with good discussions about the thesis work.

I am also grateful for the collaborations I had during my Ph.D., especially with the researchers at Los Alamos Nathan DeBardeleben, Sean Blanchard, and Heather M. Quinn.

Many thanks to my colleagues of the Group of Parallel and Distributed Processing (GPPD), for the help that goes beyond technical discussions.

I would also like to thank my family, especially my wife Natalia and my parents Delio and Mary Leusa. I could not have done it without them. Finally, I thank God who brought me to the existence and helped me all this way.

ABSTRACT

HPC device's reliability is one of the major concerns for supercomputers today and for the next generation. In fact, the high number of devices in large data centers makes the probability of having at least a device corrupted to be very high. In this work, we first evaluate the problem by performing radiation experiments. The data from the experiments give us realistic error rate of HPC devices. Moreover, we evaluate a representative set of algorithms deriving general insights of parallel algorithms and programming approaches reliability.

To understand better the problem, we propose a novel methodology to go beyond the quantification of the problem. We qualify the error by evaluating the criticality of each corrupted execution through a dedicated set of metrics. We show that, as long as imprecise computing is concerned, the simple mismatch detection is not sufficient to evaluate and compare the radiation sensitivity of HPC devices and algorithms. Our analysis quantifies and qualifies radiation effects on applications' output correlating the number of corrupted elements with their spatial locality. We also provide the mean relative error (dataset-wise) to evaluate radiation-induced error magnitude.

Furthermore, we designed a homemade fault-injector, CAROL-FI, to understand further the problem by collecting information using fault injection campaigns that is not possible through radiation experiments. We inject different fault models to analyze the sensitivity of given applications. We show that portions of applications can be graded by different criticalities. Mitigation techniques can then be relaxed or hardened based on the criticality of the particular portions.

This work also evaluates the reliability behaviors of six different architectures, ranging from HPC devices to embedded ones, with the aim to isolate code- and architecture-dependent behaviors. For this evaluation, we present and discuss radiation experiments that cover a total of more than 352,000 years of natural exposure and fault-injection analysis based on a total of more than 120,000 injections.

Finally, Error-Correcting Code, Algorithm-Based Fault Tolerance, and Duplication With Comparison hardening strategies are presented and evaluated on HPC devices through radiation experiments. We present and compare both the reliability improvement and imposed overhead of the selected hardening solutions. Then, we propose and analyze the impact of selective hardening for HPC algorithms. We perform fault-injection campaigns to identify the most critical source code variables and present how to select the best can-

didates to maximize the reliability/overhead ratio.

Keywords: HPC. Fault Tolerance. Accelerators. Radiation Experiments. Fault Injection. Reliability. Hardening Strategies. Selective Hardening.

Estratégias de Enrobustecimento para Aplicações PAD

RESUMO

A confiabilidade de dispositivos de Processamentos de Alto Desempenho (PAD) é uma das principais preocupações dos supercomputadores hoje e para a próxima geração. De fato, o alto número de dispositivos em grandes centros de dados faz com que a probabilidade de ter pelo menos um dispositivo corrompido seja muito alta. Neste trabalho, primeiro avaliamos o problema realizando experimentos de radiação. Os dados dos experimentos nos dão uma taxa de erro realista de dispositivos PAD. Além disso, avaliamos um conjunto representativo de algoritmos que derivam entendimentos gerais de algoritmos paralelos e a confiabilidade de abordagens de programação.

Para entender melhor o problema, propomos uma nova metodologia para ir além da quantificação do problema. Qualificamos o erro avaliando a importância de cada execução corrompida por meio de um conjunto dedicado de métricas. Mostramos que em relação a computação imprecisa, a simples detecção de incompatibilidade não é suficiente para avaliar e comparar a sensibilidade à radiação de dispositivos e algoritmos PAD. Nossa análise quantifica e qualifica os efeitos da radiação na saída das aplicações, correlacionando o número de elementos corrompidos com sua localidade espacial. Também fornecemos o erro relativo médio (em nível do conjunto de dados) para avaliar a magnitude do erro induzido pela radiação.

Além disso, desenvolvemos um injetor de falhas, CAROL-FI, para entender melhor o problema coletando informações usando campanhas de injeção de falhas, o que não é possível através de experimentos de radiação. Injetamos diferentes modelos de falha para analisar a sensitividade de determinadas aplicações. Mostramos que partes de aplicações podem ser classificadas com diferentes criticalidades. As técnicas de mitigação podem então ser relaxadas ou enrobustecidas com base na criticalidade de partes específicas da aplicação.

Este trabalho também avalia a confiabilidade de seis arquiteturas diferentes, variando de dispositivos PAD a embarcados, com o objetivo de isolar comportamentos dependentes de código e arquitetura. Para esta avaliação, apresentamos e discutimos experimentos de radiação que abrangem um total de mais de 352.000 anos de exposição natural e análise de injeção de falhas com base em um total de mais de 120.000 injeções.

Por fim, as estratégias de ECC, ABFT e de duplicação com comparação são apresentadas

e avaliadas em dispositivos PAD por meio de experimentos de radiação. Apresentamos e comparamos a melhoria da confiabilidade e a sobrecarga imposta das soluções de enrobustecimento selecionadas. Em seguida, propomos e analisamos o impacto do enrobustecimento seletivo para algoritmos de PAD. Realizamos campanhas de injeção de falhas para identificar as variáveis de código-fonte mais críticas e apresentamos como selecionar os melhores candidatos para maximizar a relação confiabilidade/sobrecarga.

Palavras-chave: PAD, Tolerância a Falhas, Aceleradores, Experimentos de Radiação, Injeção de Falhas, Confiabilidade, Estratégias de Enrobustecimento, Enrobustecimento Seletivo.

LIST OF ABBREVIATIONS AND ACRONYMS

ABFT	Algorithm-Based Fault Tolerance
AMR	Automatic Mesh Refinement
AVF	Architectural Vulnerability Factor
BFI	Branch Free Intervals
BID	Branch Free Interval Identifier
BSSC	Block Signature Self-Checking
CCA	Control Flow Checking Using Assertions
CFCSS	Control Flow Checking by Software Signatures
CFID	Control Flow Identifier
CI	Comparison Instructions
COTS	Commercial off-the-shelf
CSM	Continuous Signature Monitoring
DUE	Detected Uncorrectable Error
DWC	Duplication With Comparison
ECC	Error-Correcting Code
ECCA	Enhanced Control-flow Checking Using Assertions
EDAC	Error Detection and Correction
EDDI	Error Detection by Duplicated Instruction
EMU	Extended Math Unit
FDM	Finite Difference Methods
FFT	Fast Fourier Transform
FIT	Failure In Time
FPU	Floating Point Unit
FWT	Fast Wavelet Transform

GPU	Graphics Processing Units
IMCI	Initial Many-Core Instructions
ISC	Implicit Signature Checking
MBU	Multiple Bit Upset
MCA	Machine Check Architecture
MEBF	Mean Executions Between Failures
MI	Master Instructions
MTBF	Mean Time Between Failure
MTrans	Matrix Transpositions
MWBF	Mean Workload Between Failure
NW	Needleman-Wunsch
OSLC	On-line Signature Learning and Checking
PTX	Parallel Thread Execution
PVF	Program Vulnerability Factor
R-S	Reed-Solomon
RF	Register File
SBB	Storeless Basic Block
SCP	Self-Checking Pair
SDC	Silent Data Corruption
SECDED	Single Error Correction Double Error Detection
SEE	Single Event Effect
SET	Single Event Transient
SEU	Single Event Upsets
SI	Shadow Instructions
SM	Streaming Multiprocessors
SWIFT	Software Implemented Fault Tolerance

TMR	Triple Modular Redundancy
ULA	Ultra Low Alpha
VPU	Vector Processing Unit
YACCA	Yet Another Control-Flow Checking using Assertions

LIST OF FIGURES

Figure 2.1 NVIDIA Kepler Memory Hierarchy..........21
Figure 2.2 Intel Xeon Phi Architecture..........23

Figure 3.1 Triple Modular Redundancy..........28
Figure 3.2 Reed-Solomon codeword..........33
Figure 3.3 ABFT Matrix Multiplication Scheme..........36

Figure 4.1 LANSCE and ISIS neutrons spectra..........39
Figure 4.2 SDC and DUE combined Mean Workload Between Failure..........48

Figure 5.1 Experimental setup at LANSCE..........57
Figure 5.2 *DGEMM* Mean relative error and Incorrect Elements..........59
Figure 5.3 *DGEMM* spatial locality and magnitude..........60
Figure 5.4 *LavaMD* Mean relative error and Incorrect Elements..........61
Figure 5.5 *LavaMD* spatial locality and magnitude..........62
Figure 5.6 *HotSpot* Mean relative error and Incorrect Elements..........63
Figure 5.7 *HotSpot* spatial locality and magnitude..........64
Figure 5.8 *CLAMR* Mean relative error and Incorrect Elements for Xeon Phi..........65
Figure 5.9 *CLAMR* Error Locality Map..........66

Figure 6.1 Outcomes of fault injections..........73
Figure 6.2 SDC's PVF of the benchmarks for the different fault models..........74
Figure 6.3 DUE's PVF of the benchmarks for the different fault models..........75
Figure 6.4 Dependence of SDC's PVF on execution time window..........76
Figure 6.5 Dependence of DUE's PVF on execution time window..........77

Figure 7.1 Radiation test setup at ChipIR..........84
Figure 7.2 Fault injections results on the KNC and Kepler..........86
Figure 7.3 Beam experiment results, organized as relative FIT rate for SDC..........88
Figure 7.4 Beam experiment results, organized as relative FIT rate for DUE..........88
Figure 7.5 Mean Workload Between Failure..........90
Figure 7.6 GEMM relative error reduction..........93
Figure 7.7 *HotSpot* relative error reduction..........94
Figure 7.8 *LavaMD* relative error reduction..........95
Figure 7.9 *FFT* relative error reduction..........96
Figure 7.10 *DGEMM* corrupted elements dispersion..........97
Figure 7.11 *HotSpot* corrupted elements dispersion..........98
Figure 7.12 *LavaMD* corrupted elements dispersion..........98
Figure 7.13 Quick sort number of corrupted elements dispersion..........99
Figure 7.14 Merge sort number of corrupted elements dispersion..........99

Figure 8.1 Algorithm-Based Fault Tolerant for matrix multiplication..........102
Figure 8.2 Algorithm-Based Fault Tolerant FFT..........104
Figure 8.3 Duplication With Comparison strategies..........106
Figure 8.4 *DGEMM* selective hardening..........115
Figure 8.5 *HotSpot* selective hardening..........116
Figure 8.6 *LavaMD* selective hardening..........117
Figure 8.7 *LUD* selective hardening..........118

LIST OF TABLES

Table 4.1 Parallel applications details and experimental results 43

Table 5.1 Classification of parallel kernels. 55
Table 5.2 Parallel kernels' details. 56

Table 7.1 Devices under test specifications summary. 83
Table 7.2 Problem size, execution time, and FIT rates for the tested codes. 85

Table 8.1 Efficiency and resiliency of the available hardening solutions for GPUs. ... 108
Table 8.2 *DGEMM* CAROL-FI results. 110
Table 8.3 *HotSpot* CAROL-FI results. 112
Table 8.4 *LavaMD* CAROL-FI results. 113
Table 8.5 *LUD* CAROL-FI results. 114

CONTENTS

1 INTRODUCTION....14
2 BACKGROUND....17
2.1 Sources of Faults....17
2.2 Radiation Effects in HPC Accelerators....19
3 RELATED WORK....25
3.1 Reliability Evaluation....25
3.2 Hardening Strategies....28
4 RELIABILITY ANALYSIS....38
4.1 Methodology....38
4.2 Experimental Results....43
4.3 Discussion....49
5 RADIATION EXPERIMENTS CRITICALITY ASSESSMENT....51
5.1 Methodology....51
5.2 Reliability and Criticality Evaluation....57
5.3 Discussion....67
6 FAULT INJECTION CRITICALITY ASSESSMENT....69
6.1 Methodology....69
6.2 Results....72
6.3 Discussion....81
7 RELIABILITY AND CRITICALITY COMPARISON....82
7.1 Methodology....82
7.2 Reliability Evaluation....85
7.3 Error Criticality Analysis....92
7.4 Discussion....99
8 HARDENING SOLUTIONS....101
8.1 Hardware-based vs Software-based Hardening....101
8.2 Selective Hardening....110
8.3 Discussion....115
9 CONCLUSION....119
9.1 Publications....120
REFERENCES....123

1 INTRODUCTION

Accelerators are extensively used nowadays to expedite calculations in large HPC centers. Intel *Xeon Phis* and NVIDIA *Kepler* GPUs, for instance, power six of the top 10 supercomputers (DONGARRA; MEUER; STROHMAIER, 2015). Tianhe-2, Cori, Trinity, and Oakforest-PACS are powered by Xeon Phis, while NVIDIA GPUs are used as accelerators in Titan and Piz Daint. The main reasons to use accelerators are their high computational capacity, low cost, reduced per-task energy consumption, and flexible development platforms. However, 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.

Nowadays, reliability is one of the major concerns not only for safety-critical but for HPC applications as well. Various sources of faults could undermine the system reliability, including environmental perturbations, software errors, manufacturing process, temperature, and voltage variations (LUTZ, 1993; LAPRIE, 1995; NICOLAIDIS, 1999). Such faults 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 (CONSTANTINESCU, 2002; SAGGESE et al., 2005; SCHROEDER; PINHEIRO; WEBER, 2011). This work focus on radiation-induced soft errors that, according to (BAUMANN, 2005), are a considerable concern in modern computing devices because, if uncorrected, may produce a failure rate that is higher than all the other error sources combined. 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 (GOMEZ et al., 2014; TIWARI et al., 2015). As we approach exascale, the resilience challenge will become even more critical due to an increase in system scale (LUCAS, 2014; SNIR et al., 2014; RESEARCH, 2016). In this scenario, a lack of understanding of HPC device resilience may lead to lower scientific productivity and significant monetary loss (SNIR et al., 2014).

For this work, we intent to *evaluate*, *understand*, and *develop mitigation strategies* for reliability issues in current and future supercomputers. To first evaluate the problem we make a thorough analysis of HPC devices radiation reliability based on analytical studies and a series of extensive accelerated beam tests. We evaluate the error rate of registers and caches of two consecutive GPU generations. Details on pattern dependence and multiple error occurrences are also provided. Then, we study a representative set

of parallel algorithms from HPC domains, correlating their characteristics and observed radiation sensitivity.

Depending on the application and circumstances, some SDCs that are satisfactory close to the correct results may be tolerated in HPC (PUENTE et al., 2014; BREUER, 2005). To consider the outputs' error severity and better *understand* the reliability issue, we evaluate how errors manifest at the application's output and measures how the error rate reduces as a function of the tolerated level of imprecision in the output. For *HotSpot*, for instance, the error rate is reduced by 85% if a 0.5% variation in the output value is acceptable.

As part of this thesis, we designed a fault-injector, named CAROL-FI, to perform a detailed analysis of the applications' vulnerabilities to transient errors. Unlike most fault-injection frameworks, CAROL-FI injections are made at the highest possible level, to identify the algorithm 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.

To *understand* deeper the reliability issue, we investigate the reliability of six computing architectures (ARM *Cortex A9*, NVIDIA *Maxwell*, *Kepler*, *Pascal*, AMD *Steamroller*, and the Intel *Knights Corner (KNC)*). We carefully select a set of eight algorithms to compare the reliability of the considered devices. Each code has peculiar characteristics regarding memory utilization, computing power, control-flow operation, etc. To highlight specific architecture behaviors that could be generalized to similar classes of algorithms.

Finally, to *develop mitigation strategies* for HPC applications we first evaluate hardware and software techniques. We studied specific and generic hardening techniques like Algorithm-Based Fault Tolerance (ABFT) and Duplication With Comparison (DWC), and then compare these techniques with hardware implemented Error Correcting Code (ECC). We showed that ECC has the weakest protection, but it can provide the best overhead if the application is not memory-bound. ABFT has better protection with slightly higher overhead. DWC has the strongest protection but an extremely high overhead. Then, using the CAROL-FI fault injector insights, we show that a selective hardening of just a few variables can achieve protection close to the full DWC but with an overhead similar do hardware implemented ECC.

This thesis is organized as follow. Chapters 2 and 3 describe the background and related work for this thesis. Chapter 4 performs an in-depth analysis of radiation

experiments to *evaluate* the problem in an HPC device. Then, chapters 5 and 6 evaluate and go beyond the quantification of the problem and qualify the experiments to better *understand* the problem. Chapter 7 broadens our understanding performing a thorough comparison of several codes and architectures regarding the reliability issue. Finally, chapter 8 evaluates and proposes mitigation strategies for HPC applications.

2 BACKGROUND

The trend pursued by hardware designers to improve computing devices performance and reduce power consumption is to employ higher-density chips, lower voltage levels, higher clock rates, and to integrate a large number of computing cores on a single chip. While these design factors yield performance enhancements, they also make hardware components less reliable, rendering modern computing platforms very prone to experience transient radiation-induced errors not only in radiation-harsh environments such as space but also at sea level (DODD et al., 2002; GASIOT; GIOT; ROCHE, 2006).

Even a single radiation-induced fault may be harmful to both the correct operation and the performance of the software. If not appropriately detected and corrected faults can be responsible for silent data corruptions leading to altered data, incorrect program executions, erroneous results, and eventually to machine crashes (BAUMANN, 2002). It is worth noting that applying the hardening solutions designed for space applications to terrestrial ones is unfeasible (costs will be prohibitive) and pointless as the error rate at sea level is lower than space one.

These issues are an actual problem for large-scale applications and have already caused severe failures with significant monetary losses. For instance, in 2000, Sun Microsystems reported that interferences of cosmic rays with cache memories were responsible for server crashes at major customer sites, including America Online, eBay, and many others (BAUMANN, 2002). Cypress Semiconductor acknowledged similar experiences (ZIEGLER; PUCHNER, 2004) who reported monthly halts in an automotive supplier factory and havoc at a large telephone company, and also by Hewlett Packard (MICHALAK et al., 2005) reporting frequent crashes of their supercomputer at the Los Alamos Neutron Science Center. A recent large-scale study on the field conducted on Google's server fleet over a period of nearly 2.5 years also observed error rates orders of magnitude higher than previously reported in laboratory conditions (SCHROEDER; PINHEIRO; WEBER, 2011).

2.1 Sources of Faults

Electric charge disturbance is the cause of faults in silicon. This charge disturbance may alter the data state of memory structures or generate an impulse in the logic circuit that will be captured by a memory structure. If the fault affects the operation or

results of the device, it is termed error. Otherwise, the fault was masked. If the device is permanently damaged, the error will be termed hard. If the error is transient, the error is termed soft. The main cause of soft error comes from energetic ions interacting with the silicon and thus generating charge disturbance (ZIEGLER et al., 1996; BAUMANN, 2005).

2.1.1 Alpha Particles

Radioactive contaminants in package/solder materials, such as uranium and thorium, generate Alpha particles. External sources of alpha particles are not a problem as these particles penetration range in silicon is less than 100 μm.

To mitigate soft errors caused by alpha particles, Integrated circuits vendors must provide materials highly purified. To a material be considered Ultra Low Alpha (ULA) impurity must be below about one part per 10 billion. ULA implies emission at or below $0.002\ \alpha/h\ cm^2$. Soft errors induced by alpha particle will probably be less than 20% of the total soft errors if the materials are ULA.

2.1.2 Low-Energy Cosmic Rays

One significant cause of ionizing particles comes from the interactions of low-energy neutrons and boron. Boron is a material that is used intensively in integrated circuits. Low-energy neutrons come from the interaction of cosmic rays with the atmosphere. The residual products of the reaction between low-energy neutrons and boron are alpha particles and lithium recoil, both can induce soft errors.

To mitigate soft errors induced by low-energy neutrons one can simply eliminate boron from the process of fabrication. Moreover, the range of ionized particles from low-energy neutrons (alpha particles and lithium recoil) have a very limited range. Therefore, to mitigate such errors, only the first layers from the silicon should be free of boron materials.

2.1.3 High-Energy Cosmic Rays

High-energy particles, especially neutrons, originated from the cosmic rays interactions with the atmosphere, is a significant cause of soft errors. The interaction of neutrons with silicon materials may result in the nucleus breaking into ions that will induce soft errors.

Unlike alpha particles and low-energy cosmic rays, high-energy neutrons cannot be mitigated with high purity materials. Shielding could be a viable option for supercomputers. However, many meters of concrete thickness will be required to reduce the neutron flux sufficiently, increasing the complexity and costs of supercomputers site.

2.2 Radiation Effects in HPC Accelerators

This thesis will focus on soft errors which cause only transient recoverable errors. Hard errors permanently damage the device and are less likely to occur (BAUMANN, 2005). Additionally, expressive research has also been done on hard errors (MEIXNER; SORIN, 2008; POWELL et al., 2009; HONG; KIM, 2015).

Soft errors outcomes are Silent Data Corruption (SDC) and Detectable Uncorrectable Error (DUE); DUEs can be divided into a crash or hang. SDC occurs when the program exit successfully, but the output is incorrect. SDCs cannot be observed by the final user without detection techniques like parity check; Detection techniques may still not be able to detect every SDC. Crash occurs when the program state is changed in such a way that it will exit unsuccessfully, like a division by zero interruption or a segmentation fault exception. Finally, hang occurs when the program enters some infinite loop and are not able to finish at all. For HPC systems, crash and hang can be easily detected by timeout functions and unresponsive machines and will not affect the final output. The affected node of HPC systems can be rebooted, and the task may resume by checkpoint techniques or restart from the beginning. The major problem for HPC systems is, then, SDCs as it cannot be easily observable and can leave the final user with significantly corrupted outputs.

The major architectures used in HPC are accelerators, which are used to get the high performance the applications need. Intel Xeon Phi and GPUs are the two most commonly used accelerators and will be described next. Tianhe-2, the second most powerful supercomputer uses $48,000$ Intel Xeon Phi accelerators while Titan, the fifth most power-

ful, uses $18,688$ K20x GPUs as accelerators. Details about these architectures are given in the next subsections.

2.2.1 NVIDIA Graphics Processing Units

Graphics Processing Units (GPU) started aiming at rendering massively parallel graphics for personal computing. In the personal computing scenario, the probability of radiation-induced errors is low, and the multimedia applications can tolerate some errors (BREUER; GUPTA; MAK, 2004). The massively parallel architecture is also suitable for accelerate scientific applications, and nowadays GPUs are widely used to accelerate different scientific applications providing performance speedups up to 15 times (LEE et al., 2010). Moreover, there are GPUs specifically build to supercomputers. Titan, for example, uses Tesla GPUs (NVIDIA, 2015d) that are built with scientific applications in mind.

2.2.1.1 SIMD Architecture

The nature of image applications, where the same operation must be applied to a large set of pixels, led the GPU to be a SIMD architecture. GPUs can execute hundreds of simultaneous operations.

The GPU is composed of hundreds of simple cores that execute the same flow of instructions on different sets of data. Simple cores are grouped into multiprocessor elements, where the tasks are scheduled. Each core of a multiprocessor executes the same instructions on different sets of data.

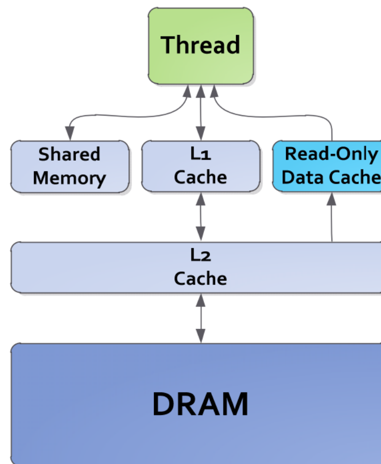
The program is structured in blocks of thread that can share data with a fast memory, called shared memory. The threads of each block can also synchronize through barriers. The blocks are scheduled to multiprocessors and execute in this multiprocessors until it finishes. Each multiprocessor can have a certain number of blocks being executed at the same time.

2.2.1.2 Memory and Cache Hierarchy

The memory of GPUs is organized into two levels, the global and shared memory. Global memory is slow but visible for all threads of the GPU. Shared memory is fast and visible only to threads in the same multiprocessor.

For the NVIDIA Kepler architecture, there are also two levels of caches. A unified L2 cache serves all the multiprocessors into the GPU. The L1 cache is private to each multiprocessor and its size is configurable using part of the shared memory. There is also an additional read-only data cache for loading constants. Figure 2.1 shows the memory hierarchy of the Kepler architecture.

Figure 2.1: NVIDIA Kepler Memory Hierarchy.



Source: NVIDIA GK110 Whitepaper[1]

2.2.1.3 Hardware Scheduler

NVIDIA GPUs hardware thread scheduler is divided into two hierarchical levels providing very fast scheduling with almost no overhead (MAITRE, 2013). The first level, called GigaThread Engine (WITTENBRINK; KILGARIFF; PRABHU, 2011), schedules $blocks$ of threads to SMs. GigaThread provides an immediate $blocks$ assignment to SMs when resources are available (PASSERAT-PALMBACH et al., 2015). The second level schedules $warps$ inside an SM. On $Kepler$ GPUs, each SM has four dual-issue $warp$ schedulers that handle four simultaneous $warps$ issuing two instructions per $warp$ per clock cycle, when possible (NVIDIA, 2015b). The four $warp$ schedulers can handle 64 simultaneous $warps$ of 32 threads or a maximum of 16 $blocks$.

2.2.1.4 Hardware Hardening and Radiation Effects in GPUs

High-End GPUs are protected only by ECC (NVIDIA, 2015d). The ECC implemented in NVIDIA GPUs can detect single and multiple errors, but only correct single

[1] Available in: https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf. Accessed 2016

errors. The resources protected are register files, shared memory, L1 and L2 caches, and main memory. Scheduler and other resources are not protected. The double-bit error detection triggers the device crash by default. ECC can be disabled leaving the device completely unprotected.

Radiation effects in GPUs can affect single or multiple threads. If a shared resource like the cache is corrupted, the radiation-induced error can affect several threads at once. Moreover, multiple bit-flips from a single particle hit can occur affecting more than one resource. The block, thread, or warp scheduler can also be hit with several threads generating SDCs, crashes or hangs. The PCI-e bus driver and hardware functions like ECC can also be hit generating SDCs or crashes. Errors affecting instructions will most likely generate a crash or hang. Therefore, a single particle hit can generate SDCs in several threads at once besides crash or hang if the resource or function executed is critical.

2.2.2 Intel Xeon Phi

Intel Xeon Phi is an Intel coprocessor developed to achieve high throughput performance. Xeon Phi aims to compete with GPUs as an accelerator for HPC systems. This architecture is similar to GPUs having many cores, targeting highly parallel applications. However, Xeon Phi features x86 in-order cores with coherent cache, supporting traditional programming models such as pthreads and OpenMP.

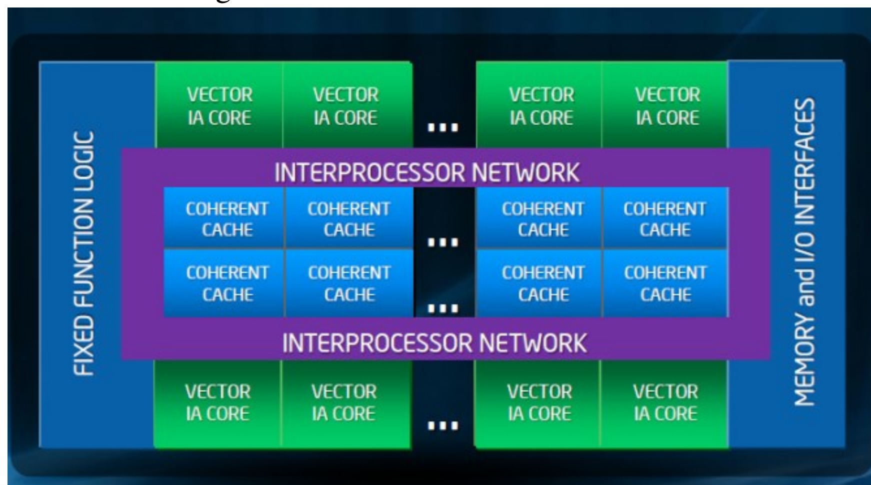
The Xeon Phi architecture is shown in Figure 2.2, it is primarily composed of processing cores, caches, memory controllers, PCIe client logic, and a very high bandwidth, bidirectional ring interconnect. Xeon Phi also supports the Hyper-Threading technology. Each core can execute up to 4 threads simultaneously, hiding memory and multi-cycle instruction latency.

2.2.2.1 Vector Processing Unit

Each Xeon Phi core has a Vector Processing Unit (VPU) and features the Intel Initial Many-Core Instructions (IMCI), that is a 512-bit SIMD Instruction set. The VPU can execute up to 16 single precision or 8 double precision operations in a single clock. Fused Multiply-Add instructions are also supported; Hence, it can perform up to 32 single

[2] Available in: https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner. Accessed 2016

Figure 2.2: Intel Xeon Phi Architecture.



Source: Intel X100 Coprocessor Architecture[2]

precision or 16 double precision operations in a single clock.

Transcendental operations such as log and square root can be executed by the VPU as it features an Extended Math Unit (EMU). The EMU unit calculates polynomial approximations of these operations.

2.2.2.2 Memory and Cache Hierarchy

Similarly to GPU, Intel Xeon Phi is mounted on a PCIe slot and has a dedicated memory. Therefore, Xeon Phi has a different memory address and communication must be done through message passing.

Two levels of cache are implemented, each core features a single cycle L1 cache divided into 32 KB L1 instruction cache and 32 KB L1 data cache and a 512 KB L2 cache. The second level cache is entirely coherent implementing a directory-based MESI coherence protocol. Xeon Phi also implements prefetch for L1 and L2 caches.

2.2.2.3 Hardware Hardening and Radiation Effects in Xeon Phi

The Xeon Phi is equipped with Machine Check Architecture (MCA), which includes various reliability solutions and logging features. Reliability solutions are used to protect memory structures and I/O operations (INTEL, 2015a). MCA covers most of the memory structures available in the Xeon Phi, but the details are intellectual property, and they are not available. MCA can correct single errors and detect some unrecoverable errors. The logging features will provide the system or user routines to react according to the errors, performing checkpoints for instance. MCA unrecoverable error detection

can be disabled to permit continued execution, instead of logging the error and triggering a crash or recovery routine. Error correction cannot be disabled leaving the device unprotected.

SDCs in Xeon Phi could be more contained than GPUs as the resources affected will be used by fewer threads than GPUs. However, errors in L2 cache caused by a single particle hit can also be spread to several threads at once. The scheduler of Xeon Phi should be less sensitive as it is managed by the operating system, with its data residing most of the time in the ECC protected main memory. Crashes and hangs will still occur as an error in critical functions, instruction, and a specific portion of the current application can lead to a crash or hang.

3 RELATED WORK

In this chapter, we first detail how to measure and evaluate the reliability of HPC applications and devices. We show all the approaches used to collect reliability data and the advantages of each one. Then, we describe the hardening mechanisms available to mitigate faults at hardware and software level. Finally, we detail the applicability of the mitigations mechanisms to HPC devices.

3.1 Reliability Evaluation

Reliability evaluation can be performed using field data providing the most realistic data. Moreover, we can also inject faults at hardware or software level to mimic realistic behaviors or to better understand the reliability issue by collecting additional information. In the following section, we detail the advantages and limitations of each approach.

3.1.1 Field Data

One approach to realistically evaluate the reliability of devices is to collect field data error logs (TIWARI et al., 2015). This approach requires access to supercomputers logs. Then, system error logs such as ECC detection and correction can be parsed and evaluated.

The time one needs to collect statistical relevant data will depend on the size and also the altitude where the supercomputer system is located. The altitude influences the radiation flux which is one of the primary sources of faults described in Section 2.1. The time span one needs to evaluate varies from a couple of months to years.

Field data, however, can only measure detectable errors such as ECC detection and system crash. Silent data corruption cannot be measured since production system cannot afford the time or energy to run the same program twice to compare outputs, or to run a code with a fixed input, without any actual result produced rather than the SDC sensitivity. Thus, there is no viable method to measure silent errors using field data.

3.1.2 Fault Injection

Since field data are not easily accessible, and can hardly provide a statistically significant amount of errors, one of the most common approaches to evaluating the reliability of devices or application reliability is to use fault injection. By injecting a fault, it is possible to calculate the Architectural Vulnerability Factor (AVF), which is the probability for corruption to propagate to the output (MUKHERJEE et al., 2003), or the Program Vulnerability Factor (PVF), which is the probability that a fault at the instruction level will affect the program output (SRIDHARAN; KAELI, 2009). If the fault injection can realistically mimic natural phenomena, like radiation beam fault injection, one can accurately measure the error rate a supercomputer is expected to experience.

3.1.2.1 Software Fault Injection

Software fault injection can be performed at different levels of abstraction from RTL to software or application level (SAGGESE et al., 2005). As the RTL level descriptions of modern accelerators are not publicly available, RTL injections could be done only on simplified circuits and, thus, may be imprecise or unrealistic (FARAZMAND; UBAL; KAELI, 2012). A higher level software fault injector is also restricted to inject faults into user-accessible resources, like register files, variables, etc. Even if it is hard for software fault injection to mimic realistic error rates and behaviors, hardware fault injections may be unpractical or too expensive. Moreover, software fault injection can provide additional information sometimes impossible to be collected with hardware fault injection, such as fault location and time of insertion.

There are several available software fault injection tools that differ in terms of injection methods and domains of application. Some examples are described below.

Ferrari (KANAWATI; KANAWATI; ABRAHAM, 1995) provides an injection mechanism based on software traps that are activated under certain conditions, like accessing a specific memory location or after a timeout. Under these conditions, a trap is activated and a fault is introduced. Ferrari can also make the fault transient or permanent depending on the user's needs.

FTAPE (TSAI; IYER, 1995) is capable of inserting faults into memory, registers, and in disk accesses. To achieve fault injection into a disk, FTAPE uses a special disk driver. Then, the data being read or written can be corrupted and delivered to the application.

SASSIFI (HARI et al., 2015) is a tool designed by NVIDIA that enables fault injection at micro architectural level on GPUs. This toll is particularly useful as GPUs are essential to many HPC and safety-critical applications. An interesting example of the latter is reliability analysis of self-driving cars. SASSIFI implements fault injection by using a low-level assembly-language tool called SASSI which profiles the code and injects faults. As SASSIFI is executed on the device, it is fast, introducing an overhead of about $5x$ the normal execution time.

GPU-Qin (FANG et al., 2014) is a fault injection tool whose goal is to obtain information regarding transient faults on GPUs. GPU-Qin provides a fault injection methodology that can achieve representative results even when considering the massive parallelism of applications which can cause simulations to last longer than acceptable simulation time. GPU-Qin is based on the debugger mode in NVIDIA GPUs. Unfortunately, as several host-device synchronizations are required to inject a fault, GPU-Qin's overhead is extremely high (up to $100x$ the normal execution time).

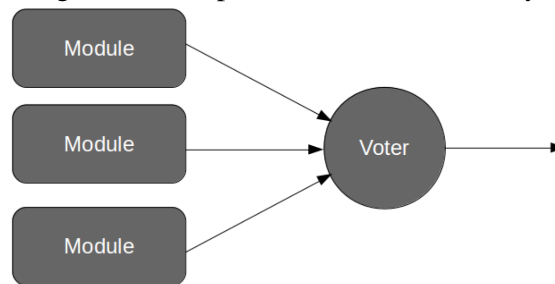
3.1.2.2 Hardware Fault Injection

Hardware fault injection usually requires extra hardware or a facility to mimic some of the events that can cause faults, such as power variation and radiation (HSUEH; TSAI; IYER, 1997). Thus, the effort to inject faults at the hardware level can be much higher, or expensive such as the need for a particle accelerator to induce radiation errors.

Methods with contact to the device attach hardware directly to the device under test to insert current or produce electrical disturbance. This method can more precisely control the place and time of fault injection. Contact methods can also mimic well permanent failures like stuck-at failures that force a permanent value in a specific place of the circuitry.

Ionizing radiation (e.g., heavy-ion or neutron beam radiation) and laser tests do not require contact, and some radiation tests mimic natural physical phenomena. Thus, such methods are the most realistic fault injection to measure error rate. However, we cannot control a specific place and time to inject a fault.

Figure 3.1: Triple Modular Redundancy.



Source: The Author

3.2 Hardening Strategies

Several hardening techniques may be applied both at hardware and software level to increase the reliability of a system. Hardware hardening includes, for instance, the enlarging of transistor capacitance, hardened memory cells, or the implementation of Triple Modular Redundancy (TMR) that typically require costly layout or architecture modifications (ZIEGLER; PUCHNER, 2004). Software hardening includes repeated code execution, either at the compiler-level or through binary code instrumentation (ZHANG et al., 2012), checkpoints, and checksum calculation (MITRA, 2012). The scope of this work is to use Commercial off-the-shelf (COTS) devices. Techniques that can only be implemented at hardware level will not be detailed.

3.2.1 Modular Redundancy

The modular redundancy is the most well-known technique to improve the reliability of hardware or software system. Modular redundancy was first envisioned by Von Neumann (NEUMANN, 1956). The most common implementation of modular redundancy is called Triple Modular Redundancy and can be illustrated by the Figure 3.1. The boxes are identical modules that can be a memory cell, logic circuit, a complete system, or any replicated module. The circle is called the majority voting circuit; This circuit takes as input the output of each replicated module and the output is the majority voter. Therefore, if one of the modules produces faulty outputs, the voter will forward the output of the other two modules masking the faulty module output.

TMR is a technique that can detect faults and mask them. Because there are more than two modules, if an error occurs, the majority voter can still be considered correct. With error masking, TMR can perform a continuous operation as the fault module output

will not be propagated and the execution will safely continue. When there are only two modules, the system is called Self-Checking Pair (SCP). It is not possible to perform a continuous operation using SCP systems, and this configuration can only detect if a fault occurs if the outputs of the identical modules do not match. When faults are detected in SCP systems, it can trigger the full re-execution, a smarter rollback strategy, or any defined error treatment strategy.

Modular redundancy was studied with the simultaneous multithreading chips leading to ideas such as partial redundant multithreading (PARASHAR; SIVASUBRAMANIAM; GURUMURTHI, 2006; MUKHERJEE; KONTZ; REINHARDT, 2002). The main idea of partial redundant multithreading is to efficiently use hardware resources and reuse data to improve performance. The key principle is to define a sphere of replication where faults in the leading or trailing thread will be detected.

The strategy proposed by Mukherjee in (MUKHERJEE; KONTZ; REINHARDT, 2002) is to set the largest possible sphere of replication including processor pipeline and register files, but not replicating L1 data and instructions caches. The hardware is modified to delay the trailing thread so that cache misses for the leading thread will be fetched before the trailing thread executes the load. Another optimization is to eliminate control flow misprediction by using the result of the leading thread branches. Additionally, Slice-based threading and value and control flow locality optimizations were included in (PARASHAR; SIVASUBRAMANIAM; GURUMURTHI, 2006) to further improve the performance.

3.2.2 Control Flow Checking by Signature Monitoring

Signature-monitoring techniques have been proposed to detect control flow errors. The main idea is to assign signatures to all Basic Blocks of the code. The signatures can be assigned arbitrarily or derived from some characteristics such as binary code or instructions addresses. During program execution, runtime signatures are generated and checked against precomputed signatures to check control flow errors. The signatures can be computed using the current and previous Basic Blocks characteristics. The scheme used to generate signatures will determine which control flow error it can detect, such as an illegal branch or branching in the middle of the Basic Block.

Many techniques propose the use of dedicated hardware to generate runtime signatures and compare to the ones calculated at compile time. Dedicated hardware is used

for: Continuous Signature Monitoring (CSM) (WILKEN; SHEN, 1991), On-line Signature Learning and Checking (OSLC) (MADEIRA; SILVA, 1992), and Implicit Signature Checking (ISC) (OHLSSON; RIMEN, 1995). Many approaches use software-only techniques to be able to use COTS hardware, such as Block Signature Self-Checking (BSSC) (MIREMADI et al., 1992), Control Flow Checking by Software Signatures (CFCSS) (OH; SHIRVANI; MCCLUSKEY, 2002a), and the Concurrent Control Flow Checking approach in (YAU; CHEN, 1980)

CFCSS were evaluated in (GOLOUBEVA et al., 2003), the memory and performance overhead range from 107% to 338% for a Sparc V8 microprocessor. The fault injection was performed inserting bit-flips only in immediate operands of the branch instructions. For the fifth order elliptical wave filter benchmarks, CFCSS increase the percentage of incorrect results. In general, CFCSS reduced the percentage of incorrect results by 7% to 86%.

3.2.3 Control Flow Checking Using Assertions

Control Flow Checking Using Assertions (CCA) divides the instructions into sets of Branch Free Intervals (BFIs), each BFIs then has two identifiers. The Branch Free Interval Identifier (BID) is unique for each BFIs; BIDs are set to a variable at the entry-point of the BFI and checked at the end of BFI. If the BID does not match, the BFI starts at a different point than the entry-point and a control-flow error is detected. Control Flow Identifier (CFID) is the second identifier and is the same among the BFIs that share the same parent BFI. CFID is checked to ensure the correct sequence of BFIs. The CFIDs are stored in a queue of size two. The CFID of the next BFI is enqueued at the beginning of the BFI. A CFID is dequeued at the end of BFI and checked against the CFID of the current BFI. If the program attempts to enqueue in a full queue, dequeues an empty queue or fails the CFID check, an error is detected.

The main problems of CCA are the high overhead and the additional branches inserted by the technique, which remains vulnerable to control flow errors. To solve the high overhead, Enhanced Control-flow Checking Using Assertions (ECCA) (ALKHALIFA et al., 1999) divides the code into blocks, which can be a collection of BFIs where there are only one entry point and one exit point. By tuning the size of the block, ECCA can reduce the number of assertions inserted in the protected code. To solve the additional control flow instructions, ECCA uses a scheme with prime number defined at preprocessing time

and one variable assignment at runtime. The assignment is devised in such a way that if a control flow occurs a division by zero will also occur, raising then the division by zero exception. The scheme can also identify if the division by zero occurred because of control flow error or a data divided by zero.

ECCA is unable to detect control flow errors that remain in the same Basic Block. Yet Another Control-Flow Checking using Assertions (YACCA) (GOLOUBEVA et al., 2003) improves the error detection capability by using signature monitoring technique. YACCA generates the signatures at compile and runtime to use in the assertions. Additionally, YACCA inserts a rule in the assertions to detect faults in the decision operand of a conditional branch.

Results in (GOLOUBEVA et al., 2003) showed a memory and performance overhead ranging from 107% to 630%. The fault injection model used inserts random bit-flips in the immediate operands of the branch instructions. The unhardened version of the codes showed correct results for 49% to 56% of the faults injected. A maximum of 25% of the faults generates incorrect results in the fault injecting campaign. When ECCA and YACCA are applied, a maximum of 4% and 1% of the faults produce incorrect results respectively.

3.2.4 Error Detection by Duplicated Instruction

The main goal of Error Detection by Duplicated Instruction (EDDI) is to detect errors introduced in the systems (OH; SHIRVANI; MCCLUSKEY, 2002b). The idea is to include duplicated instructions into the code. The original instructions are called Master Instructions (MI), and the duplicated ones are Shadow Instructions (SI). The general purpose registers, as well as memory, are also duplicated for the use of MI and SI. Comparison Instructions (CI) are also included to compare registers and memory values from both partitions, original and duplicated, to detect an error. If a mismatch is detected by CIs, an error handling function will be invoked.

Consider the following instruction:

```
ADD   R3,   R1,   R2   ; R3 ← R1 + R2
```

The code will then be transformed in the following MI, SI, and CI:

The register R1, R2, and R3 are the master registers while R21, R22, and R23 are the shadow ones. As the values of master and shadow registers should be the same, the

```
ADD  R3,   R1,   R2         ; MI
ADD  R23,  R21,  R22        ; SI
BNE  R3,   R23,  gotoError  ; CI
```

result stored in R3 and R23 should also be the same. Therefore, the CI is inserted using a conditional branch to compare the values of R3 and R23.

The overhead of SI and CIs can be lowered as the results that need to be checked are only the ones before store instructions or conditional branches. Therefore, CIs will only be inserted after the end of blocks of computation; the authors call such blocks as Storeless Basic Block (SBB). Each SBB is a store and branch-free sequence of instructions; then a CI is inserted before the store or branch instruction at the end of each SBB.

The results in (OH; SHIRVANI; MCCLUSKEY, 2002b) show a performance overhead ranging from 13% to 111% for the eight benchmarks executed in a 4-way superscalar R10000 MIPS processor. Without the EDDI protection, 30% to 60% of the faults injected did not produce incorrect results. The high percentage of correct results despite the faults inserted is because the value corrupted was masked or never used. Moreover, from 6% to 38% of the faults injected produced incorrect results without EDDI. Using EDDI, only a maximum of 2% for the benchmarks tested produced incorrect results.

3.2.5 Error Detection and Correction

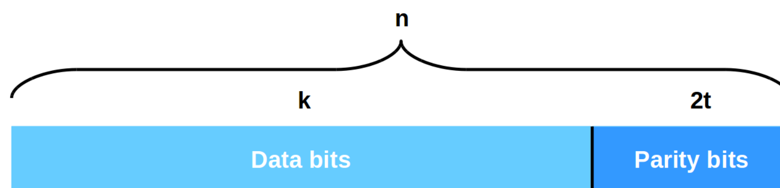
Error Detection and Correction (EDAC) (LABEL et al., 1996) can be used to detect and even correct a limited number of bit-flips in a sequence of binary data. The first example of an EDAC method is the parity check bit. The parity check is a detect only method where the idea is to simply count the number of logic ones in the binary data. Then, the extra parity bit is set to indicate if there is an odd or even number of logic ones. This parity check technique can detect if there were an odd number of bit-flips, but is unable to detect an even number of bit-flips.

Hamming code is another technique that can detect single or two bit-flips and can correct single bit-flips (MOON, 2005). A simple Hamming code inserts r redundancy bits to n data bits such that $2^r \geq n + r + 1$. For example, it needs 7 bits to protect 64 bits of data. The redundant bits are then positioned inside the data bit word in a 2^n bit pattern. For example, for a 7-bit word with four redundant bit, the positions of redundant bits are 1, 2, 4, and 8. The final word can be defined as $PPBPBBBPBBB$ where B is data bit, and P is parity bit. If only a single bit-flip occurred, the error pattern of parity bits could

be used to identify the erroneous bit position and corrected it.

Another EDAC method is called Reed-Solomon (R-S) coding (WICKER, 1994). This R-S method can detect and correct multiple and consecutive bit-flips. R-S is constructed by using k data symbols of s bits each. With $2t$ parity symbols, the final codeword will consist of n data symbols such that $n = k + 2t$. The example is depicted in the Figure 3.2.

Figure 3.2: Reed-Solomon codeword.



Source: The Author

Decoding the codeword using finite field arithmetic, R-S can detect $2t$ erroneous symbols and correct t erroneous symbols. One common example of R-S is the examples called (255,233) where $n = 255$, $k = 223$, and the symbols are 8-bit size, then, $2t = 36$ and $t = 16$. For (255,233), R-S can correct 16 8-bit erroneous symbols, each symbol can have one erroneous bit or all of them.

3.2.6 SWIFT

Software Implemented Fault Tolerance (SWIFT) (REIS et al., 2005b) is a technique which applies compiler transformations to duplicate instructions and insert comparisons at code generation. Basically, the idea is to use EDDI, ECC, and Control Flow Checking techniques in conjunction and apply some optimizations. SWIFT is a software-based technique that requires main memory to be hardware protected by ECC, then, it cannot be applied to any system.

The first optimization comes from the conjunction use of EDDI and main memory with ECC protection. Considering the main memory protected, EDDI duplicates only values outside of the main memory. Then, using EDDI and ECC will reduce the number of load and store instructions by half, improving performance. The memory footprint will also be reduced, as the main memory will not be duplicated.

SWIFT insert Control Flow Check as EDDI will detect data errors, but control flow errors can still occur. Another optimization is that control flow correctness can be checked only in Basic Blocks that have store instructions. The erroneous value will only

leave the sphere of replication, and cannot be detected afterward when store instructions are executed. SWIFT, then, update the signature in all Basic Blocks but checks only in Basic Blocks with a store. This late check reduces the number of instructions and increases the performance while the fault detection will remain the same.

There are two main point-of-failures in this approach. The first one is when the data is corrupted between the value validation and store instruction. If the data value or address is corrupted when the store is executed, the error will not be detected. The second one is when the opcode of instruction is changed to a store, this store will be executed without validation and can corrupt values out of the sphere of replication.

Fault injection was performed using PIN to modify register values in (REIS et al., 2005b). Unprotected code finished 15% of executions with an error while SWIFT had no finished executions with error, detecting 70% of the faults. 63% of the executions finished successfully for the unprotected code, and 18% for the SWIFT protected code. The high number of successful execution of unprotected code is because most of the faults were masked, not affecting the final output. The low number of correct executions for SWIFT is because unharming faults, that not affect the final output, is detected nevertheless and the execution is marked as faulty. The rest of the executions crashed with a segmentation fault error.

3.2.7 Checkpoint and Rollback

The checkpoint is a common strategy used from large scale to reliable systems (MITRA, 2012; NAKSINEHABOON et al., 2008; EGWUTUOHA et al., 2013; BAUTISTA-GOMEZ et al., 2011; MARUYAMA; NUKADA; MATSUOKA, 2010). The main idea is to save the program state to a safe memory, creating checkpoints of correctly known states. When an error is detected, a rollback strategy is then applied to return the program to a correct state and resume the execution from there. This technique can be applied to recover from crashes and easily detected errors. However, if some fault detection mechanism is available, checkpoint and rollback can also be applied to recover from faults that could generate incorrect results and would otherwise be assumed to be correct.

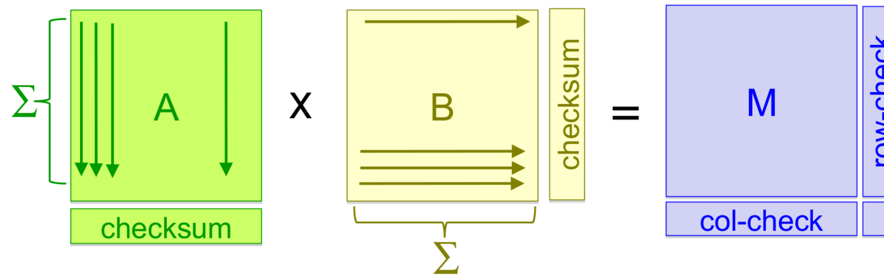
The main drawbacks of the checkpoint are the memory overhead, used to save the program state, and the performance overhead as the program needs to be paused during the checkpoint creation. To better use this strategy, the user needs to evaluate the error frequency to define the best checkpoint interval.

3.2.8 Algorithm Specific Techniques for Fault Tolerance

Algorithms resilient to silent faults have been developed so far in a variety of clean but often unrealistic models. Among others, we remind: the liar model (PELC, 2002) which assumes transient comparator failures but no corruption of data, the models for the design of resilient data structures residing in large and unreliable memories in a checking scenario (BLUM et al., 1994; CHU; KANNAN; MCGREGOR, 2007) and in a recovery setting (AUMANN; BENDER, 1996), the adversarial faulty-RAM model (FINOCCHI; ITALIANO, 2008), where a malicious adversary can arbitrarily corrupt at any time a fixed number of memory cells. In particular, many resilient solutions for a variety of fundamental algorithmic problems (such as sorting (FINOCCHI; ITALIANO, 2008) and suffix trees (CHRISTIANO; DEMAINE; KISHORE, 2011)) have been studied in the faulty-RAM. The connection between fault tolerance and I/O-efficiency has been preliminarily investigated in (BRODAL; JØRGENSEN; MØLHAVE, 2009). Many significant problems are still unsolved in faulty computational models: most notably, no resilient data structures for storing graphs are known in the literature, and even basic graph traversal algorithms are very poorly understood. Algorithm engineering work on resilient algorithms is also in a very early stage, and only a few experimental papers contribute carefully engineered implementations (see, e.g., (FERRARO-PETRILLO; FINOCCHI; ITALIANO, 2009)).

All these works focus on sequential models of computation. However, the work of Abraham in (HUANG; ABRAHAM, 1984; JOU; ABRAHAM, 1988), which is called Algorithm-Based Fault Tolerance (ABFT), can be directly applied to parallel algorithms. For matrix operations, Abraham showed that the checksum of the input matrices, A and B, will preserve some characteristics after certain matrix operations such as multiplication. For example, in a matrix multiplication where $M = A \times B$, transforming the input matrices A and B to include the checksums of each column and row respectively, we can use these values in the resulting M matrix to detect and correct some errors. This matrix transformation is depicted in the Figure 3.3. The Fast Fourier Transform can be hardened in a similar way as demonstrated in (JOU; ABRAHAM, 1988).

Figure 3.3: ABFT Matrix Multiplication Scheme.



Source: (RECH et al., 2013a)

3.2.9 Hardening Techniques Applicability to HPC Devices

The techniques presented in the previous subsections are among the best options to include in HPC devices as software-based technique. There are some variations of the techniques which were not described here, but the key ideas presented in the techniques here are sufficient to give an overall good insight on how to improve the resilience of HPC devices.

Modular redundancy can be used to achieve high levels of reliability. However, modular redundancy can substantially increase the costs and complexity of the design at the hardware level. Software modular redundancy can be easily implemented eliminating the hardware cost and complexity, but the execution time overhead will significantly increase. However, limiting the sphere of replication, duplicating only a few portions of the code can be a good strategy if the code portions are wisely chosen. Later in section 8.1.3, we show the results of software modular replication applied in HPC devices.

EDDI technique is very similar to modular replication. One of the benefits of using this instead of modular replication would be to lower the scheduling stress, which can be beneficial if the architecture has a sensitive scheduler, as is the case of GPUs (RECH et al., 2013b). However, Modular redundancy and EDDI cannot be indiscriminately applied as the performance overhead rapidly increases resulting in poor performance, which can be unacceptable to HPC.

Control flow checking can be used to detect errors faster than other techniques, such as modular replication, that would wait for the entire execution to finish and check the outputs. Therefore, control flow check could speed up the recovery time from faults. However, this technique will not suffice if the scientific applications are mainly data flow, and many of the scientific applications are engineered to be mainly data flow as the devices are designed to be more efficient this way. The nature of HPC applications that executes

in an accelerator, then, limits the gain that could be obtained by control flow strategies.

EDAC techniques can be efficiently implemented in HPC devices (CURRY et al., 2008). Software-based implementation can be used to verify the consistency of data produced by an application. However, the main drawback would be to change the algorithm to use the extra bits necessary to check and correct the output data. EDAC, then, could lead to increased costs and time to write a scientific application with performance overhead and resilience gains that can vary widely according to the specific EDAC used.

The checkpoint technique may face problems such as different memory address space from HPC accelerators and host server. Another issue is the cost to copy the memory back to accelerators and restart the application, inserting an extra overhead. However, this technique can also be applied, and some accelerator like Xeon Phi already implement some checkpoint (INTEL, 2015a) that can be used.

SWIFT technique introduces the idea to bring together techniques to best suit the hardware and software. SWIFT also presents the idea to harden only portions of the code to lower the overhead. The results were expressive, and HPC devices could benefit from such approach.

Algorithm specific fault tolerance techniques can yield great results, even when using COTS hardware. This work will not focus on a specific algorithm or class of algorithms, but in a more general way to improve the resilience of several applications. However, well known and well used algorithms, like matrix operations, are used inside bigger applications and the ABFT hardening can still be used to improve the overall resilience of such applications.

The available hardening techniques can then be applied to HPC devices with particular advantages for the algorithm and hardware used. Therefore, well understanding the device and algorithm, we can match the best hardening techniques or some variation of them to produce the best results for HPC applications and devices.

4 RELIABILITY ANALYSIS

In this section, a reliability analysis using radiation experiments for a representative set of parallel algorithms for HPC applications is presented. Usually, when performing radiation experiments, the error propagation observability is limited in a way that is not possible to identify the source of the observed failure unequivocally. As a result, the evaluation may be limited to the tested configuration and be hardly generalizable. The aim is to go a step further with the experiments by selecting a broad set of applications and configurations that cover a wide range of computational and data movement requirements.

4.1 Methodology

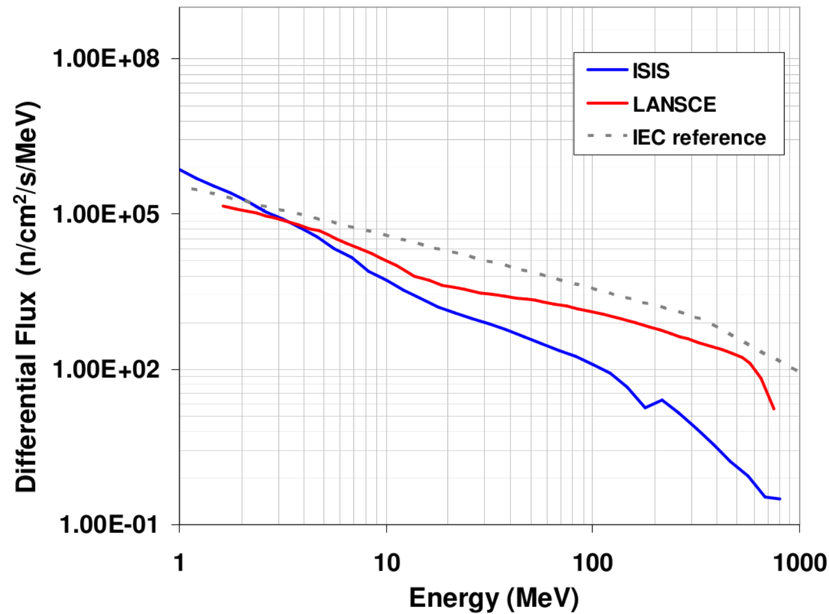
The natural flux of heavy ions that wanders in the deep space, protons that surround the earth and fast neutrons created by the incidence of cosmic rays on the earth's atmosphere can be simulated through the use of particle accelerator facilities. Such facilities attempt to mimic the energy spectrum of particles, but with a flux that is millions of times greater than the terrestrial one, one hour of a test at these facilities represents many hundreds of years of natural exposure. This accelerated flux, in turn, allows performing extensive testing to assess the sensitivity of electronic devices to radiation.

The radiation experiments presented in this work were performed at the Los Alamos National Laboratory (LANL) Los Alamos Neutron Science Center (LANSCE) Irradiation of Chips and Electronics House II, and at the VESUVIO beam line in ISIS, Rutherford Appleton Laboratories, Didcot, UK. Figure 4.1 shows that LANSCE and ISIS provide a white neutron source that emulates the energy spectrum of the atmospheric neutron flux from 10 to 750 MeV. Moreover, the beam at ISIS and LANSCE was empirically demonstrated to mimic the terrestrial radiation environment (VIOLANTE et al., 2007).

The neutron flux used in both facilities was higher than the neutron flux at sea level (JEDEC, 2006). However, it is important to notice that is very unlikely to have more than one neutron hit inducing an error per execution in normal condition, then, we carefully designed the experiments to make sure that the probability of more than one neutron generating a failure is negligible. The observed error rates were lower than 10^{-2} errors/execution.

The beam was focused on a spot with a diameter of 2 inches, which provided uniform irradiation of the device chip without directly affecting nearby board power control

Figure 4.1: LANSCE and ISIS neutrons spectra plotted against the reference of neutrons spectrum at the sea level.



Source: (VIOLANTE et al., 2007)

circuitry and DRAM chips. The beam focus implies that data stored in the main memory is not corrupted, allowing an analysis focused on just the device cores. Actually, having the DRAM exposed would have masked most of the device cores behavior under radiation that was intended to highlight. Moreover, DRAM sensitivity to radiation has already been deeply analyzed and proved to decrease with the shrinking of technology nodes (BAUMANN, 2005), and modern DRAM chips are provided with efficient ECC circuits that increase device reliability by several orders of magnitude (KIM et al., 2007). It is worth noting that such a consideration does not apply to caches, for which technology shrinking, compact design, and performance requirements increase the probability of having failures as well as the efforts and penalties of adding ECC (ASADI et al., 2005).

A host computer initializes the test sending pre-selected input to the accelerator and gathers results, comparing them with a pre-computed golden output. When a mismatched is detected, the execution is marked as affected by a *Silent Data Corruption (SDC)*. To avoid precision and round-off issues, golden outputs were calculated on the very same device used for experiments. Input values were ensured to be small enough to avoid overflow but still big enough to be considered representative. Additionally, to avoid biases on input values, small input sizes are a subset of big input sizes and input has been generated balancing the number of 0s and 1s.

Copying memory from the host computer to the devices through the PCIe bus is a very time consuming operation, even longer than the code execution time itself when

working with accelerators. During the memory copy, the device is in idle mode and DRAM is not irradiated. Thus, the time spent during memory copy is wasted and should be reduced at the minimum. The input vector is copied from the host to the device DRAM only at the beginning of each run, and those inputs are used for several code executions. Once an error is detected, DRAM content is dumped and checked to ensure input consistency. Observed SDCs were never caused by input errors (DRAM is not irradiated). As SDC rate is relatively low (about one error every 10^3 executions), this methodology smooths memory copy overhead and improve the effective device exposure time.

Software and hardware watchdogs were included in the setup. The software watchdog monitors the application under test detecting *DUEs*, i.e., application crashes or control flow errors that prevent the device from completing assigned tasks (e.g., the device enters an infinite loop). The hardware watchdog is an Ethernet controlled switch that performs a power cycle of the host computer if the host computer itself does not acknowledge any ping requests in ten minutes. The hardware watchdog is necessary as radiation can corrupt the PCIe controller on the device board as well, possibly causing the host computer to hang.

4.1.1 Metrics to Evaluate Experimental Results

Radiation experiments aim at measuring the *cross section*, which is the sensitive area of the device. i.e., that portion of area that, if hit by an impinging particle, causes an observable failure. When an algorithm is tested, the cross section is measured dividing the observed error rate ($errors/s$) by the average particle flux ($particles/(cm^2 \times s)$), yielding an area. The larger the cross section, the higher the use of sensitive resources (BAUMANN, 2005; MUKHERJEE et al., 2003). So, the cross section depends on the overall amount of resources required for computation and on their criticality, but does not include any information on execution time. As ISIS and LANSCE reasonably mimic the atmospheric neutron flux, the probability of having a neutron corrupting the device during our experiments or at sea level during normal operation is very similar (VIOLANTE et al., 2007). The experimentally observed cross section is then an intrinsic characteristic of the device and code, independent on the neutron source. Multiplying the cross section (cm^2) with the expected neutron flux on the device ($13n/(cm^2 \times h)$ at sea level (JEDEC, 2006)), one can estimate the device error rate or Failure In Time (FIT), expressed as $errors/10^9h$.

4.1.2 Device Under Test

In this chapter, we consider NVIDIA *K20* which was the accelerator used in the Titan supercomputer at Oak Ridge National Laboratory. Titan was the second fastest supercomputer at the time measurements were performed.

Kepler K20 GPUs are designed by NVIDIA in a 28nm technology node (NVIDIA, 2015d). The K20 is composed of 13 SM, each of which is divided into 192 CUDA cores. K20 features a 706MHz SM core clock, 1.25MB L2 cache, a total of 832KB in L1 cache, and a total of 3.25MB of register file storage. Register files, shared-memory, L1 and L2 caches are SECDED protected, read-only data cache is parity protected. The tested devices have CUDA capability 3.5, which allows each SM to execute a warp of up to 192 parallel threads in a single computing cycle. If the block size exceeds 192, the execution of some threads will be delayed until the computation of the preceding warps of the block has been completed. Each SM has two schedulers. At every instruction issue time, the first scheduler issues one instruction for some warp with an odd ID and the second scheduler issues one instruction for those with an even ID, when double-precision floating-point instructions have to be executed, the second scheduler cannot issue any instruction.

The experiments with K20 were conducted with the ECC mechanism disabled. On the NVIDIA GPUs ECC can be disabled using $nvidia - smi$ tool (NVIDIA, 2015d). Performing the experiments with reliability mechanisms enabled would require a large amount of time to gather a statistically significant amount of data. Additionally, while disabling mitigation mechanisms seems unrealistic, disabling them is fundamental to find the raw architectures reliability. ECC, for instance, could mislead SDC and DUE distinction. In fact, when the ECC is disabled, a double bit error may be masked without affecting the code output (WILKENING et al., 2014). The same double bit error will trigger an application DUE when ECC is ON (NVIDIA, 2015d). Later in this thesis, we also evaluate the efficiency and efficacy of ECC in section 8.1.4, we discovered that ECC reduces the SDC rate by about one order of magnitude.

4.1.3 Selected Algorithms

Several benchmark suites are available for performance and efficiency evaluation of computer architectures (BAILEY et al., 1994; WOO et al., 1995; BIENIA et al., 2008; CHE et al., 2009). A standard set of benchmarks for the reliability evaluation of HPC

devices has not been established, yet. General guidelines for reliability evaluation of computing devices suggest to consider codes from different domains and comprising different computation and communication patterns (ASANOVIC et al., 2006; QUINN et al., 2015). Therefore, we selected algorithms representative of different domains and application classes. The algorithms are detailed in the following.

Matrix Transpose is a procedure that copies a complete matrix to another swapping columns and lines element by element. Matrix Transpose is a representative tool for various graphical procedures that do not make any operation on data, but reorders it. Such data transfers are common on matrix rotation and many matrix transformations. The Matrix Transpose workload can also be considered to be similar to sorting algorithms, where elements are not modified but only rearranged.

Matrix Multiplication serves as a cornerstone kernel for several applications and performance evaluation tools. The simple implementation, MxM, uses one thread to compute each cell of the resulting matrix. The *DGEMM* implementation reuses a thread to compute sixteen cells of the output matrix and uses local memory and registers to maintain parts of the source matrices. Memory accesses are coalesced/vectorized. This strategy results in a better memory locality and a high device utilization, but also stresses the register file, local memory, and Floating Point Unit (FPU).

Fast Fourier Transform is one of the most representative algorithms in HPC. *FFT* algorithms are used in several applications such as signal processing, vibration and spectrum analysis, speech processing, communication, linear algebra, statistics, and stock options pricing determination (KRüGER; WESTERMANN, 2003; OWENS et al., 2008). *FFT* is based on the code developed by Volkov and Kazian (VOLKOV; DEMMEL, 2008). Each 512-point 1D-FFT is computed by a block of 64 threads to improve global memory access and use shared memory.

LavaMD calculates particle potential and relocation due to mutual forces between particles within a large 3D space. This space is divided into cubes, or large boxes, that are allocated to individual blocks of threads (CHE et al., 2009). The main computation in this kernel lies on dot products with floating-point data, where each thread computes the interaction of one particle with all particles in neighboring boxes (26 neighbor boxes in the cutoff radius plus the home box allocated to the block of threads). As the home box and a neighbor box are kept at all times in local memory, and each particle's data includes coordinates and velocity, *LavaMD* stresses local memory the most.

HotSpot simulates the energy dissipation on an architectural floor plan to estimate

processor temperature (CHE et al., 2009). At each iteration, *HotSpot* computes the average temperature on areas of the chip based on their previous temperature and power input. The kernel behaves like a 2D stencil computing on single-precision floating-point values. Given *HotSpot* small local memory footprint, a high number of iterations using local memory and registers only, and use of single-precision instead of double-precision, *HotSpot* achieves the highest occupancy among tested codes.

Needleman-Wunsch was one of the first applications of dynamic programming used to compare biological sequences (NEEDLEMAN S.B., 1969). *NW*, developed by Saul B. Needleman and Christian and D. Wunsch is an important algorithm in bioinformatics to align protein or nucleotide sequences. *Needleman-Wunsch* is based on the comparison of integer values only, therefore, no floating point operations are computed. *NW* organizes a 2D matrix in groups of 32×32 cells, parallelism happens between cells in the antidiagonal as a wavefront. The kernel is called each time to compute independent groups of cells on the same antidiagonal.

4.2 Experimental Results

Table 4.1: Parallel applications details and experimental results. †Tested in LANSCE, §tested in ISIS

	Input ($\approx$)	Output ($\approx$)	Instr. Exec. ($\approx$)	Ex. time $[s]$	SDC FIT	DUE FIT
$MxM(2^{10})^\dagger$	$2.10M$	$1.05M$	$1.07G$	0.05	4.63×10^2	3.97×10^2
$MxM(2^{11})^\dagger$	$8.39M$	$4.19M$	$8.59G$	0.37	5.79×10^2	2.64×10^2
$MxM(2^{12})^\dagger$	$33.55M$	$16.78M$	$68.72G$	2.90	6.03×10^2	2.61×10^2
DGEMM$(2^{10})^\dagger$	$2.10M$	$1.05M$	$1.07G$	0.01	7.43×10^2	2.79×10^2
DGEMM$(2^{11})^\dagger$	$8.39M$	$4.19M$	$8.60G$	0.02	7.83×10^2	1.96×10^2
DGEMM$(2^{12})^\dagger$	$33.55M$	$16.78M$	$68.75G$	0.14	1.04×10^3	2.28×10^2
$MTrans^\S$	$4.19M$	$4.19M$	$88.08M$	3.46	1.80×10^1	1.60×10^1
$FFT^\dagger$	$33.55M$	$33.55M$	$402.65M$	0.07	2.88×10^3	7.02×10^2
$LavaMD^\S$	$1.69M$	$1.69M$	$63.49G$	0.18	3.44×10^3	1.52×10^3
$Hotspot^\S$	$2.10M$	$1.05M$	$14.68G$	7.49	2.04×10^2	1.12×10^2
$NW(2^{12})^\dagger$	$8.19K$	$16.78M$	$65.56M$	0.06	6.33×10^1	5.48×10^1
$NW(2^{13})^\dagger$	$16.38K$	$67.10M$	$261.73M$	0.22	3.06×10^2	2.17×10^2
$NW(2^{14})^\dagger$	$32.77K$	$268.44M$	$1.05G$	0.83	9.00×10^2	3.60×10^2

To give an overview of the difference among the tested algorithms (described in section 4.1.3), Table 4.1 lists the amount of input and output data, intended as double-precision floating-point elements for all the algorithms but for NW, which is executed with integer data. Each complex element in FFT is counted as 2 double numbers. Please

note that *DGEMM* requires an additional input matrix with respect to MxM that was filled with zeros in our experiments, which is why *DGEMM* and MxM are listed with the same input size.

It is reasonable to believe that the higher the amount of data to be elaborated, the higher the error rate. Nevertheless, the radiation sensitivity may not be linearly dependent on input size. In fact, computational power will also influence the radiation sensitivity of the code. A higher number of instructions, threads, and operations will increase the probability for a neutron to generate a control flow error, a scheduler failure, and a single transient event that propagates to the output (RECH et al., 2014). MxM, *DGEMM*, and NW were tested with different input sizes to study the radiation sensitivity dependencies on problem size. When increasing the input size, only the number of instantiated blocks was increased, while the number of threads per block was kept constant, which allows maintaining the SM caches and resources efficiency throughout the different configurations.

Table 4.1 also lists the number of CUDA instructions each benchmark executes and the kernel execution time. $LavaMD$ is the benchmarks with the highest computational demand as it needs one order of magnitude more $instructions - per - data$ (obtained dividing column 5 by column 3 in Table 4.1) than the others. $MTrans$, the opposite case, is considered to evaluate the radiation effects on a GPU when rapid data movements are performed with little computation.

The last two columns of Table 4.1 show the results of experimental evaluation for all the tested algorithms. $LavaMD$, $Hotspot$, and $MTrans$ were tested at ISIS in December 2013 and May 2014 while the others were tested at LANSCE in September 2013 and December 2014. FIT are reported with a 95% confidence interval that includes both statistical error and neutron counts uncertainty.

4.2.1 Silent Data Corruption

The SDC rate ranges from 1.80×10^1 FIT for $MTrans$ to 3.44×10^3 for $LavaMD$, varying by almost two orders of magnitude between tested applications (Please, refer to Table 4.1). The underlying reason is differences in the GPU resources utilization and intrinsic code characteristics (i.e., the sensitivity of resources (MUKHERJEE et al., 2003)), amount of data elaborated, and the number of executions performed. For instance, the sensitivity of register files for the $Hotspot$ and MxM benchmarks has been estimated

to be approximately 20% while for the $MTrans$ benchmark the estimation is one order of magnitude lower (about 2%) (TAN et al., 2011). As register files represent more than 50% of memory resources in the K20, it is reasonable to assume that registers are among the main sources of errors. Our experimental observations agree with the sensitivity estimations as the difference of one order of magnitude is preserved for both SDC and DUE rates. Fast Wavelet Transform (FWT) also shows a sensitivity of 20% for register files (TAN et al., 2011). As FWT is comparable with FFT, we can conclude that a similar sensitivity applies to FFT register files, which agrees with our FFT FIT rate trend. Applications that heavily use register files are then expected to be more prone to be corrupted.

The difference between MxM and *DGEMM* is of particular interest as they solve the same problem with different implementations, modifying the resources criticality (mainly memory). Data in the GPU memory is exposed to radiation and susceptible to be corrupted. No effect is expected if the corrupted data has already been digested by the GPU, is obsolete, or will be overwritten. Data that still has to be used or needs to be written back is critical and, if corrupted, leads to an observable failure which is counted in the FIT measure. In other words, a high caches hit rate brings better performances but a higher SDC cross section and FIT.

In MxM, the data required by a thread does not fit in the caches, so the execution time is dominated by memory latencies to move data from the DDR. While waiting for new data, possible neutron-induced failures in logic are not critical as the thread is in idle state, and data in cache is likely to be removed to accommodate new requests, so radiation corrupts obsolete data. Meanwhile, a thread in *DGEMM* digests all data in the SM caches before requiring new elements, and the whole optimization aims at maximizing the hit rate in cache and reducing accesses to the device memory. Under radiation, this turns into a higher criticality for *DGEMM* caches, which increases the code FIT. The resource utilization efficiency, then, affects the radiation sensitivity of the GPU. State registers, on the contrary, are critical during memory exchange latencies, which is why MxM has a higher DUE rate than *DGEMM*, as described in the following subsection.

4.2.2 Detectable Uncorrectable Error

The DUE rate depends on the application, and it ranges from between 20% and 30% of the SDC rate for *DGEMM* to 55% for $Hotspot$, and almost 90% for $MTrans$ and

NW(see the last two columns of Table 4.1). The higher occurrences of DUE vs. SDC in $MTrans$ are explained noting that DUEs are directly related to the number of instructions executed by a thread that, if corrupted, lead to a control flow error. Additionally, it depends on the number of parallel threads instantiated (i.e., the scheduler strain required for computation). In $MTrans$, it is unlikely for radiation to corrupt data, as it sits on internal memory for a very short period of time. $MTrans$ does not perform arithmetical operations on data but only re-allocations. A failure in such an operation may have a high probability of leading to a control flow error and, thus, to a DUE. NW has little data to work on, but a great number of operations are performed on each element. NW will then require a higher amount of instruction cache than data cache with respect to other algorithms (DANALIS et al., 2010). A failure in the instruction cache is likely to generate a control flow error, explaining why for NW the ratio of DUE vs. SDC is higher than other codes.

4.2.3 Input Size Variation

To further investigate the operative behaviors of GPUs under radiation, the amount of data elaborated alone is not sufficient to indicate the program neutron-induced error rate. For example, $MTrans$ elaborates $4\times$ more data than $LavaMD$ but has a two orders of magnitude lower SDC FIT. Similarly, FFT elaborates about $20\times$ more data than $LavaMD$, but has a comparable SDC FIT. The SDC sensitivity of a code actually increases with input dimensions, as demonstrated for MxM, ***DGEMM***, and NW executed with different input sizes, but the increase is not linear. In MxM and ***DGEMM***, the GPU caches are fully loaded even with the smallest tested size, so the actual exposed sensitive area does not change linearly with the input size. If a cache region is used twice for computation, the data it holds changes but the exposed area remains the same. The same occurs for logic computing resources. Parallel codes for GPUs are typically extremely regular (DANALIS et al., 2010; NVIDIA, 2014). When the input size is increased, there is no significant modification in the number and kind of per-data-operations to be performed. If the GPU is fully loaded, the increased input size requires the reuse of some logic gates without a significant increase in the GPU exposed area. As a result, the cross section and FIT increase with input size are caused by the additional resources required to manage a higher amount of blocks (the number of threads per block is kept constant) and synchronize a more complex execution (RECH et al., 2014). In NW, the FIT increase

with input dimension is more remarkable, as the GPU memories are not fully filled. The input integer vectors, in fact, largely fit in the K20 register files, 1,280KB L2 and 832KB L1 caches. Thus, increasing the input dimensions has the side effect of increasing the GPU used (and exposed) area. As a general result, increasing the amount of data elaborated increases the GPU SDC FIT. If the data fills the GPU memory, the increase is sub-linear and caused by the additional scheduling resources required.

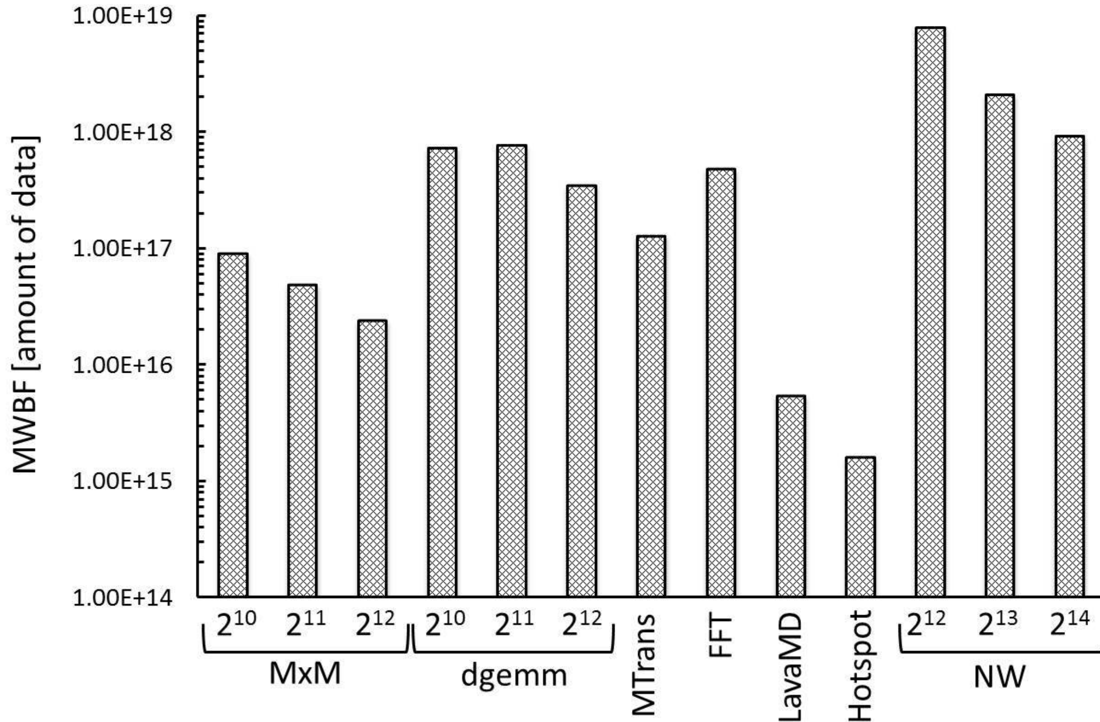
When the problem size is increased, the number of instructions in the instruction cache is not significantly increased, as the code is compiled only once and executed with different inputs (KIRK; WEN-MEI, 2012). It is unlikely for CUDA to introduce input specific optimizations at kernel launch or run time. This is in accordance with experimental data reported in the last column of Table 4.1 that shows an almost constant DUE FIT for MxM, *DGEMM*, and NW when the problem size is increased (variations are inside statistical error tolerance). The combination of SDC increase and almost constant DUE makes the ratio of DUE vs. SDC to be significantly reduced with increasing problem size. In fact, DUEs are 85% of SDC for $MxM(2^{10})$ and 43% for $MxM(2^{12})$. A similar trend is observed for *DGEMM* and NW. As a general result, we can conclude that increasing the problem size will not significantly increase the DUE FIT of the application.

4.2.4 Code Comparison

The code radiation sensitivity discussed in the previous subsection indicates the probability for a neutron that strikes the GPU during the code execution to generate an observable failure. When analyzing processors reliability, the execution time has also to be taken into account. The execution time, in fact, determines the number of neutrons that hit the processor during computation. The higher the execution time, the more neutrons will impinge the processor before completing computation. To take execution time into account, the *Mean Executions Between Failures* (MEBF), defined as the number of executions correctly completed between failures, was introduced (WEAVER et al., 2004). The MEBF is calculated dividing the Mean Time Between Failures (MTBF) by the code execution time.

On a GPU, and parallel processors in general, all the design and programming efforts aim at increasing performance. Not taking execution time into account would then result in an unfair and imprecise GPU reliability evaluation. Moreover, the GPU throughput depends on the input size, so the amount of data elaborated is not linearly dependent

Figure 4.2: SDC and DUE combined Mean Workload Between Failure.



Source: The Author

on the execution time. The codes *Mean Workload Between Failure* (MWBF), measured multiplying the MEBF by the amount of data elaborated in each execution (RECH et al., 2014) (double-precision floating-point elements for all the codes except NW, which works with integers) need to be evaluated. The MWBF depends on the used resources reliability and the resource efficiency like the MEBF, but also on the throughput gain the resources bring to the code. Basically, the MWBF will also consider the fact that GPU throughput varies with input size.

Figure 4.2 shows the MWBF for the tested algorithms and versions, considering either SDC or DUE as failures. From the figure, it is clear that fewer data can be correctly computed in $LavaMD$ and $Hotspot$ with respect to other codes. This is expected, as those are the codes with the highest instructions-per-data and the lowest throughput.

Even though *DGEMM* has a higher FIT than MxM, Figure 4.2 shows that much more data can be correctly elaborated by the former than the latter. For matrix multiplications, the compute-bound implementation brings a reduction in the execution time which is enough to compensate the increase in radiation sensitivity. *DGEMM* is then the implementation to be chosen, as it combines the best performance with the highest reliability. Even if this result is not generalizable, the use of MWBF allows a more precise evaluation of GPU reliability.

MxM and NW MWBF decreases with input size. For MxM, this is caused by the increased scheduling resources required (and thus the increased FIT) and its reduced efficiency. When the input size is increased, each thread will execute more operations that require more data, thus increasing waiting time for memory and reducing the GPU throughput. NW throughput does not decrease with input size (DANALIS et al., 2010). However, the excessive increase in the cross section documented in the previous subsection undermines the benefit that the throughput possibly brings. Choosing *DGEMM* with size $2^{11} \times 2^{11}$ seems to be the most reliable solution for matrix multiplication. *DGEMM* MWBF increases slightly from 2^{10} to 2^{11} indicating that *DGEMM* throughput efficiency and reliability is improved when increasing the input size. At $2^{11} \times 2^{11}$, the maximum throughput gain is reached as the caches and register utilization saturates and does not improve the throughput for bigger input sizes. As a general programming advice, increasing the workload of a parallel code increases reliability if the throughput gain is sufficient to balance the higher FIT rate caused by the higher use of scheduling resources.

4.3 Discussion

The reliability of scientific algorithms in HPC devices is a significant problem as the results show. Highly efficient algorithms may provide a better MWBF as is the case do *DGEMM* wich suits best the GPUs architecture. However, different algorithms classes, like $LavaMD$ and $HotSpot$, show a very different result. Moreover, the algorithms need to improve the resilience, especially if we consider the future exascale supercomputers. Today supercomputers already have an MTBF of dozen of hours (TIWARI et al., 2015), and exascale supercomputers will suffer an even worse scenario if we do not improve the reliability of hardware and software.

To appropriately and efficiently use today and future HPC supercomputers, we need to devise fault tolerance strategies that can provide high resilience without undermining the performance gain of such architectures. One of the possible ways to improve the resilience of such algorithms is to implement software-based hardening techniques. The hardening techniques need to take into account the algorithm characteristics to use the best opportunities to improve resilience. Therefore, as each algorithm behave differently to radiation-induced errors, we need to thoroughly understand such errors for specific algorithm classes and provide techniques that best suit the algorithm and hardware to balance the performance/resilience trade-off. Then, a methodology will be needed

to better assess the algorithms and hardware to provide sufficient information to wisely improve the resilience.

5 RADIATION EXPERIMENTS CRITICALITY ASSESSMENT

In this chapter, we aim to not only have a quantitative evaluation of radiation-induced errors, but a qualitative evaluation to better understand the reliability issue. Thus, we propose a novel error criticality evaluation methodology based on specific metrics. We then evaluate two HPC devices used by the first two fastest supercomputers according to the Top500 list at the time of measurements. With the proposed methodology, we identify which architecture is more likely to produce critical errors in the tested codes and which parallelism management philosophy is more reliable than the other.

5.1 Methodology

In this section, we detail the metrics proposed to perform the qualitative evaluation and the devices under test. Then, we present the selected algorithms and the reason for the chosen input sizes. Finally, we describe the experimental procedure adopted for this evaluation.

5.1.1 Metrics

We select four metrics to characterize radiation-induced output errors and to discuss their criticality in HPC applications: the number of incorrect elements, relative error, mean relative error, and spatial locality.

We design our radiation experiments to have at most one neutron generating a failure per execution, as detailed in section 4.1. When multiple elements in the output are corrupted, it means that the effect of that single impinging neutron propagates and spreads affecting multiple processes or values. The higher the **number of incorrect elements** in the output data, the more likely for a code to propagate the error and exacerbate the number of incorrect elements in the output. The number of incorrect elements, then, correlates well with the algorithm and architecture sensitivities.

The number of incorrect elements is especially significant for parallel architectures. HPC devices like Intel Xeon Phi and NVIDIA GPUs have dozens or thousands of cores that share different levels of resources. If a shared resource is corrupted, several threads may produce incorrect data. Moreover, each device handles parallelism differ-

ently. NVIDIA has simple cores and a hardware scheduler (NVIDIA, 2015b; NVIDIA, 2015a) while Intel uses more complex cores and a complete operating system with software scheduler (INTEL, 2015b; INTEL, 2015a). The corruption of the scheduler or operating system is likely to affect the execution of multiple threads.

To measure output error magnitude, we calculate the **relative error** which is given by the following equation:

$$relative\,error = \frac{|read - expected|}{|expected|} \times 100.$$

Where *read* is the value of the corrupted element and *expected* is the correct one. Relative error is a measure of how off the corrupted result is from what is expected, expressed in percentage. The relative error of a corrupted element that has a value which is ten times the expected will be 900%. If an algorithm produces text as output, one could apply **relative error** treating the output as an integer.

The **mean of relative errors** is obtained averaging the relative errors of all the corrupted elements in the output. The mean of relative errors gives an overview of how much the overall corrupted output differs from the expected one. In our analysis, we correlate the mean of relative errors with the number of incorrect elements. This correlation highlights how many elements were corrupted and how much those elements differ from the expected value. We can then distinguish situations in which radiation produces few corrupted elements that are significantly different from the expected value, and situations in which the corruption affects many elements which are only slightly different from the expected value.

We also use the relative error to filter those errors that significantly impact the results and those errors that could be ignored. Some applications may accept as correct results that slightly wander off the precise value. For instance, a seismic wave application accepts misfits of about 4% (PUENTE et al., 2014). Additionally, the relative error becomes fundamental for imprecise computations (SIMONITE, 2016; ERNST et al., 2004; BREUER, 2005). In this work, being conservative, we chose to consider only mismatches with relative errors greater than 2%. We are aware that the accuracy or relative error allowed by a scientific application or imprecise computations may vary widely. Hence, we made available all our corrupted outputs in a publicly accessible repository (RECH et al., 2016) so to allow users to apply different filters.

When we apply the filter, we ignore all incorrect elements whose relative error is lower than 2%. We remove faulty executions where there are no mismatches left after the

filter. As will be shown in Section 5.2, several errors could have a relative error inferior to some parameterized threshold. Considering every mismatch as an error would be, then, an ineffective evaluation of resilience and error criticality. Therefore, the reliability of architectures and algorithms that produce low relative errors could be immensely far from the real one.

It is common for HPC output data to be structured as two or three-dimensional arrays. The **spatial locality** of errors, then, identifies the output errors pattern. When several elements are corrupted, but they do not share the same position in one of the axis, they are tagged as random errors. When the corrupted elements share one, two, or three dimensions of the axis we classify them as line, square, or cubic respectively. The spatial locality of errors is important to understand error propagation in the considered architecture and how data is actually used in the device. Locality information can be fruitfully used to evaluate software-based hardening strategies detection efficacy. For example, the Algorithm-Based Fault Tolerance (ABFT) *DGEMM* can detect and correct single and line errors (HUANG; ABRAHAM, 1984; RECH et al., 2013a) but not square errors. Therefore, by knowing the spatial locality we can evaluate if it is wise to implement ABFT. It is worth noting that the spatial locality can be deeply affected by the relative error, as the number of incorrect elements can decrease as we apply the filter.

The four presented metrics can be used in conjunction to better understand the reliability of an algorithm or architecture and conceive a solution to improve their resilience. The number of incorrect elements in the output can indicate the magnitude of error propagation. Correlating the number of incorrect elements with the mean relative error provides an overview of output correctness. Locality can give insights on errors propagation and help to understand data placement or organization. Locality could also contribute to the development and use of detection and correction strategies (BERROCAL et al., 2016).

5.1.2 Devices Under Test

For this criticality evaluation, we consider two devices: NVIDIA K40 and Intel Xeon Phi 3120A. The K40 is an updated version of the K20 used in the previous chapter and is widely used in supercomputers in the Top500 list. Xeon Phi is a device introduced by Intel to accelerate performance in supercomputers similarly to NVIDIA GPUs, however with the advantage to using X86 codes. Intel Xeon Phi has also been adopted by supercomputers listed in the Top500 and is used by Tianhe-2 in China, which was the

fastest supercomputer at the time measurements were performed. Thus, we evaluate two state-of-the-art HPC devices used by supercomputers.

The K40 includes a Kepler architecture based on the GK110b GPU chip (NVIDIA, 2015c). The GK110b is fabricated using 28nm planar bulk technology from TSMC and includes 15 Streaming Multiprocessors (SMs), up to 2048 threads/SM, 30 Mbit total register file (RF), 960 KB total L1 cache/Shared memory (64 KB per SM), 1536 KB L2 cache, and 12 GB GDDR5 (which is not irradiated).

The Xeon Phi board, codenamed Knights Corner (KNC), is the coprocessor 3120A (INTEL, 2015b; INTEL, 2015a). The coprocessor is fabricated using 22nm with the Intel 3-D Trigate transistors. The chip includes 57 physical in-order cores with four hardware threads and 32 512-wide vector registers per core. The board has 6GB GDDR5 (which is not irradiated) with 64 KB L1 cache and 512 KB private L2 cache for each core (a total of 3648 KB and 29184 KB for L1 and L2 caches, respectively). L2 caches are fully coherent and connected using a bidirectional 64 bytes wide data ring.

The physical implementations of Intel and NVIDIA devices are extremely different. 3-D transistors have shown a 10$\times$ reduced per bit sensitivity to neutron compared to planar devices (NOH et al., 2015). The raw resources corruption probability for the Xeon Phi is then expected to be lower than for the K40. Unfortunately, as circuit level details are proprietary, it is not possible to evaluate the devices low-level resources sensitivity. A direct comparison between NVIDIA and Intel devices physical implementation reliability is then unfeasible and out of the scope of this work. We focus on the criticality of radiation-induced error, which depends on how the error propagates till the application output and is related to the device architecture.

NVIDIA's and Intel's management of parallel processes are extremely different and may impact both the device efficiency and reliability. NVIDIA has a hardware scheduler while Intel relies on a dedicated Operating System (OS) to orchestrate execution. The characterization of the parallel threads management is part of the goal of our test procedure (details in Section 5.1.4).

5.1.3 Selected Algorithms

To select a representative set of algorithms we choose: ***DGEMM***, ***LavaMD***, ***HotSpot*** detailed in section 4.1.3. We also include ***CLAMR***, which is a Department of Energy (DOE) mini-application (GUAN et al., 2015). *CLAMR* is a DOE homemade fluid dynamics ap-

Table 5.1: Classification of parallel kernels.

	Bound by	Load Balance	Memory Access
DGEMM	CPU	Balanced	Regular
LavaMD	Memory	Imbalanced	Regular
HotSpot	Memory	Balanced	Regular
CLAMR	CPU	Imbalanced	Irregular

plication, representative of classified LANL supercomputers workloads. *CLAMR* simulates the long range propagation of waves using a cell-based adaptive mesh refinement implementation. By using the shallow water equations (conservation of mass, x momentum, and y momentum) and by assuming that the fluid bottom is flat and that the flow in the vertical direction is negligible, the simulation is implemented by having each cell of the 2D space computed by a thread. *CLAMR* stresses FPU resources (by being compute-bound and working over double-precision floating-point data), control flow resources (the kernel uses several tests to handle questions like border conditions), and device control resources due to its large number of kernel calls and changes in number of threads between time steps to re-balance the load among computational resources.

We believe that results obtained with the selected benchmarks could be, under certain premises, generalized to similar applications. It is worth noting that we should restrain experimental radiation evaluation to few benchmarks because of beam time limitations and the need to gather a statistically significant amount of data.

To broaden the representativeness of the selected applications, we have classified each code using some general parameters such as: resources bounding the execution (i.e., either CPU or memory), load balance (balanced or imbalanced), and the regularity of the memory access pattern – which affects the capacity of the algorithm to profit from the memory hierarchy (e.g., coalesced accesses). Table 5.1 shows the classification for the selected applications.

It is worth noting that even if the high level code of the selected algorithms is the same for both devices, the post compiler code may be very different between the K40 and the Xeon Phi. The code difference is due to different architectures and compilers. Nevertheless, as highlighted earlier in the section, the selected set of codes is heterogeneous in the sense that each stimulates a particular kind of resources the most. To reach the solution, both Xeon Phi and K40 devices are forced to use those resources.

Table 5.2: Parallel kernels' details.

	Domain	Input size	#Threads
DGEMM	Linear algebra	square matrix side ($2^{10} - 2^{13}$)	side$^2/16$
LavaMD	Molecular dynamics	grid size ($13, 15, 29, 23$)	grid size$^3\times$ #particles (100 on Xeon Phi, 192 on K40)
HotSpot	Physics simulation	#cells (1024×1024)	#cells
CLAMR	Fluid dynamics	#cells (512×512)	#cells or more (AMR)

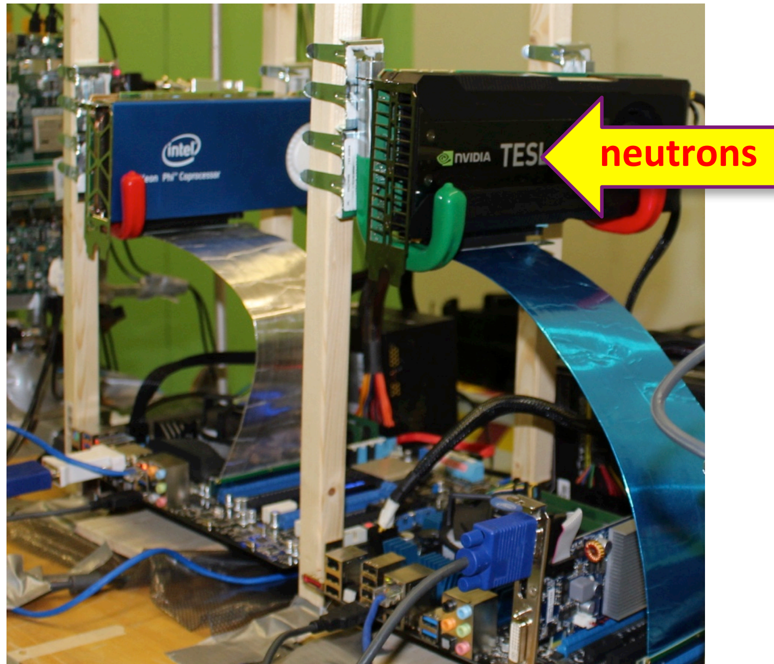
5.1.4 Selected Input Sizes

To have a proper reliability evaluation, it is essential to fully utilize the device resources. An underused device can give different error criticalities due to smaller resource usage and fewer threads created. Input sizes were tailored to achieve high resource utilization (e.g., over 97.5% multiprocessor activity on the K40). This includes register files, cache memories, buses, ALUs, FPUs, control resources, and others. Table 5.2 resumes the input size and number of threads generated for each kernel and the selected configuration to achieve high resource usage. *DGEMM* input sizes (cell per matrix side) were varied between $2^{10} \times 2^{10}$ and $2^{13} \times 2^{13}$ in powers of two. *LavaMD*'s number of cubes in each dimension of a 3D grid was set to 13, 15, 19, and 23 (each cube contains 100 particles on Xeon Phi and 192 particles on K40. The number of particles was selected to best fit the hardware).

As tested input sizes are sufficient to saturate most of the resources on both devices, a bigger input size does not increase the number of resources required for computation and should not affect FIT (BAUMANN, 2005). However, increasing the input size increases the number of instantiated parallel processes, and modifies the shared resources distributions. Moreover, for most HPC applications the throughput is strongly dependent on the input size. Evaluating how error criticality changes with input size provides novel insights on parallel processes management reliability.

HotSpot's 2D stencil includes 1024×1024 cells. The workload employed in *CLAMR* is the standard test problem of a circular dam break. The mesh starts with 512×512 cells and simulates $5,000$ timesteps.

Figure 5.1: Part of the experimental setup at LANSCE. Neutrons direction is indicated by the arrow.



Source: The Author

5.1.5 Neutron Beam Test Experimental Setup

Experiments were performed at the LANSCE facility, Los Alamos, NM, and at the ISIS facility, RAL, Didcot, UK. Figure 5.1 shows part of the experimental setup mounted at LANSCE. We irradiate a total of 2 Xeon Phis and 2 K40s, placed at different distances from the neutron source. A de-rating factor was applied to consider distance attenuation. After the de-rating the device radiation sensitivity seemed independent on the position, suggesting that the neutron attenuation caused by other boards between the source and the device under test is negligible. The experimental setup is described in detail in section 4.1.

5.2 Reliability and Criticality Evaluation

This section evaluates error criticality of HPC application classes. The analysis is based on the metrics proposed in Section 5.1.1 using the codes and methodology presented in Section 5.1. While this work focuses only on SDCs, with our methodology it is also possible to measure radiation-induced DUEs. As a reference, we measured that SDCs are between 1.1 to tens of times more likely than DUEs for both the K40 and Xeon Phi. For *DGEMM*, K40 experienced between $1.1\times$ to $4\times$ more SDCs than DUEs (the

larger the input, the higher the DUE rate), while for Xeon Phi SDCs are about $4\times$ more likely than DUEs (independently on the input). For *LavaMD*, K40 has about $3\times$ and Xeon Phi from $3\times$ to $12\times$ (increasing with input size) more SDCs than DUEs. For *HotSpot*, K40 has $7\times$ and Xeon Phi has $3\times$ more SDCs. Observed differences may be dependent on algorithm control-flow characteristics, control logic sensitivity, instruction cache proprieties, or architecture peculiarities. We consider DUEs less critical than SDCs as, for their nature, they are detectable. A detailed analysis of DUEs causes and effects is then out of the scope of this work. In the following, we consider only SDCs obtained during our radiation experiments.

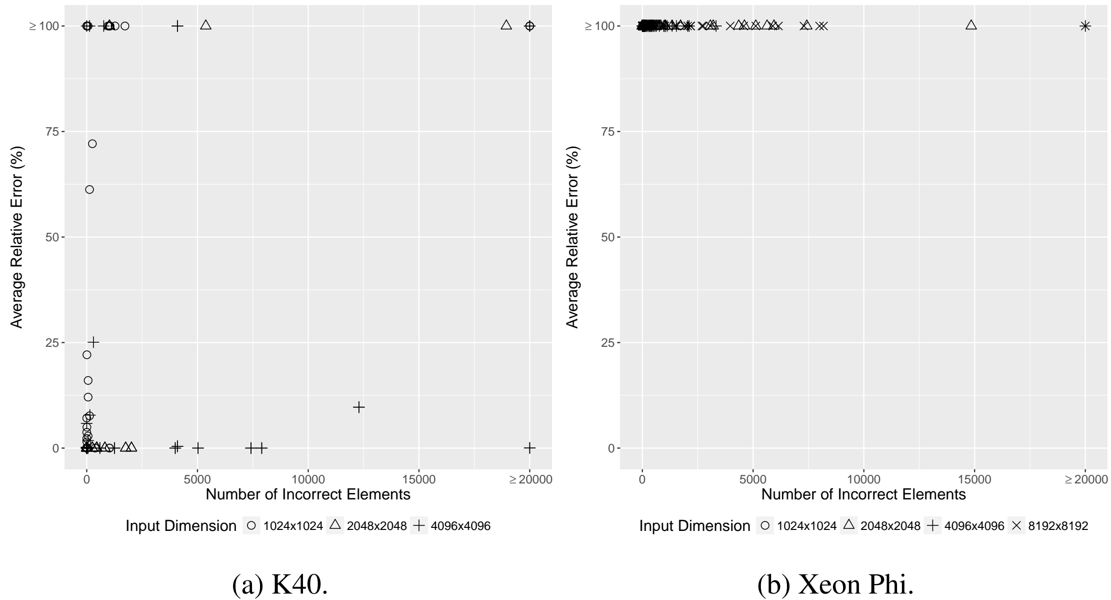
Results are presented as relative FIT, expressed in arbitrary units ($a.u.$). Absolute FITs are considered business-sensitive data and are not included in this work to protect our industrial partners. Nevertheless, as we use the same normalization for each device and code, relative FIT data still allows cross comparisons between codes and devices. As stated in Section 5.1.2, Xeon Phi and K40 have extremely different architectures built with different transistor layouts. The scope of this work is not to exhaustively compare the error rate of the two devices, but rather to evaluate and compare the corrupted output criticality for different classes of algorithms with different input sizes executed in different HPC devices.

5.2.1 *DGEMM*

Figures 5.2a and 5.2b show the mean relative error correlated with the number of corrupted elements for the faulty executions of *DGEMM* executed with 3 and 4 input sizes on K40 and Xeon Phi, respectively. It is worth noting that, to improve figure quality, for *DGEMM* we assign a 100% relative error to all those errors with a relative error higher or equal to 100%.

Most executions had a small number of incorrect elements in both architectures (at most 0.4% of the output elements corrupted). The number of incorrect elements grows together with input size. We recall that, as described in Section 5.1.5, observed (multiple) corrupted elements are caused by a single impinging particle. When multiple corrupted elements affect the output it means that the initial corruption propagates disturbing the calculation of more than one element. An increase of *DGEMM* input size requires a higher number of parallel processes and a higher amount of shared resources (like caches). A corruption in either one is likely to cause multiple corrupted elements.

Figure 5.2: *DGEMM* Mean relative error and Incorrect Elements.



(a) K40. (b) Xeon Phi.

Source: The Author

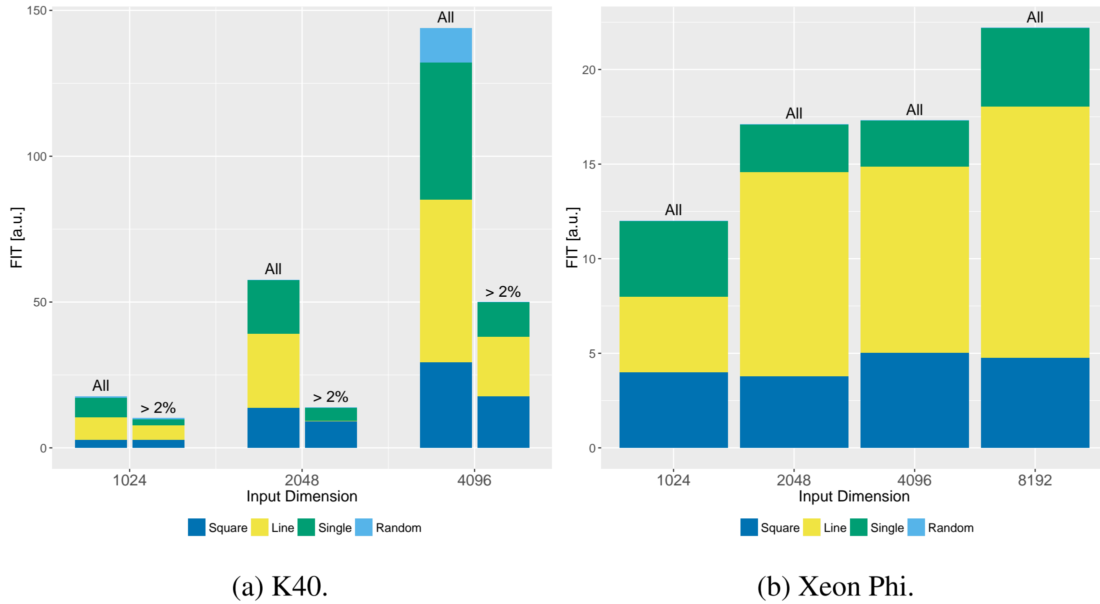
As shown in Fig. 5.2b, the mean relative error is extremely high on the Xeon Phi. Almost all the corrupted elements are extremely different from the expected value, independent of the number of corrupted elements or input dimension.

For the K40, about 75% of radiation-induced output errors have a lower than 10% mean relative error. The K40 has overall fewer corrupted elements and those elements' values are less different to the expected ones than on the Xeon Phi, indicating that *DGEMM* errors are then to be considered less critical on the K40 than on the Xeon Phi.

Figures 5.3a and 5.3b present the spatial locality and relative errors for *DGEMM* executed on K40 and Xeon Phi. For each input size, we show the relative FIT break down into the different error patterns detected with our spatial locality analysis. For each dimension we report two FIT break downs, one considering all the corrupted executions and one applying the 2% relative error filter (All and $> 2\%$, respectively, in Figure 5.3a). For the $> 2\%$ break down, we do not consider as corrupted those output elements with a relative error lower than 2%. As for the Xeon Phi no relative error was lower than 2%, we present only the FIT break down for all errors. For the K40, on the contrary, 50% to 75% corrupted executions had all the elements with a relative error lower than 2%. Therefore, if we tolerate 2% of discrepancy from the correct value, K40's reliability is at least 60% better than when considering all mismatches.

It is worth noting that for the K40 errors distribution changes when results are fil-

Figure 5.3: *DGEMM* spatial locality and magnitude.



(a) K40.

(b) Xeon Phi.

Source: The Author

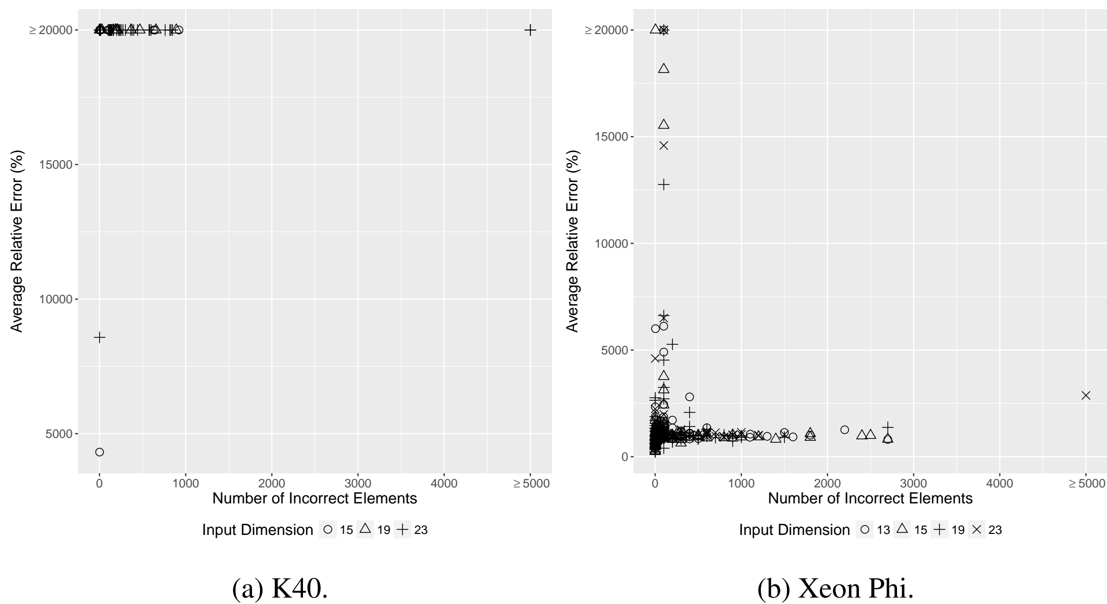
tered with the 2% tolerance. Random distributed errors almost disappear while single and line errors are significantly lowered. The 2% filter does not clear those incorrect elements with a magnitude higher than 2%. One execution classified as square may change to line or single when some elements are filtered. Unfortunately, the spatial distribution after the filter depends on the magnitude of each incorrect element and cannot be easily predicted.

Spatial locality has a strong impact on the effectiveness of hardening strategies like ABFT (HUANG; ABRAHAM, 1984). Single and line are easily corrected in linear time on parallel devices (RECH et al., 2013a; WUNDERLICH; BRAUN; HALDER, 2013) while square and random errors are more difficult to detect and correct. Therefore, applying ABFT, *DGEMM* would be affected by only 20% to 40% of all errors on K40, and 60% to 80% on Xeon Phi.

Even if an exhaustive comparison between K40 and Xeon Phi is out of the scope of this work, comparing Figures 5.3a and 5.3b it is clear that even considering a 2% tolerance in the output, the K40 has still a higher error rate than the Xeon Phi. If ABFT is applied to both devices, the error rates become comparable.

It is interesting to notice that the input size has a strong impact on K40 FIT but not on Xeon Phi FIT. From $2^9 \times 2^9$ to $2^{11} \times 2^{11}$ K40 FIT increases of $7\times$ for ALL and $5\times$ for $> 2\%$ while Xeon Phi FIT increases of only $1.8\times$. As discussed in Section 5.1.4, the different behavior between NVIDIA and Intel devices when input size is increased depends mainly on two reasons that derive from the different parallel threads management

Figure 5.4: *LavaMD* Mean relative error and Incorrect Elements.



(a) K40.

(b) Xeon Phi.

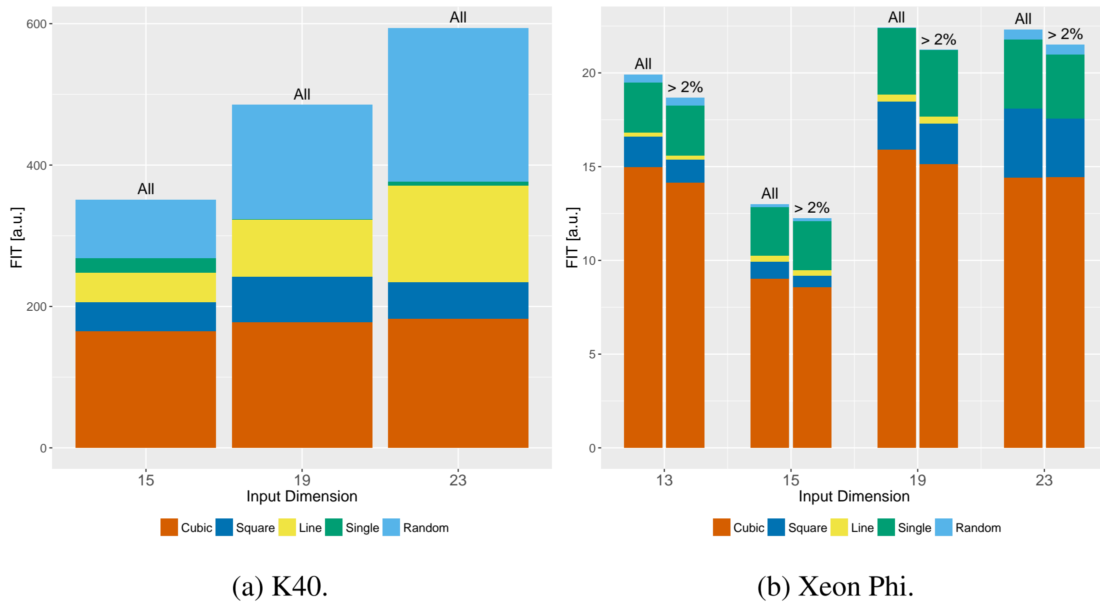
Source: The Author

philosophies:

(1) Increasing the number of parallel threads increases the scheduler strain required to manage and dispatch threads. The scheduler on NVIDIA devices is implemented in hardware and has already been demonstrated to contribute to the device radiation sensitivity (RECH et al., 2014). Intel Xeon Phi relies on the operating system to manage execution (INTEL, 2015a) which may be less susceptible to radiation-induced failures. It is worth noting that while the K40 thread management seems to increase its sensitivity, it may be more efficient. The K40 may then produce more correct data before experiencing a failure (RECH et al., 2014).

(2) NVIDIA and Intel adopt opposite solutions to manage those threads that are active but waiting to be dispatched. On the K40, active threads' data is kept in registers while other threads are being executed. A larger number of threads increases, then, the time data stays exposed in registers waiting to be used. The available ECC on K40 registers mitigates this effect, but data may still sit in internal queues or flip-flops that are not protected. On the contrary Xeon Phi waits for current threads (up to four per core) to finish before launching other ones. Subsequent threads' data sit in the DRAM, so there is no expected FIT increase caused by additional threads.

Figure 5.5: *LavaMD* spatial locality and magnitude.



(a) K40.

(b) Xeon Phi.

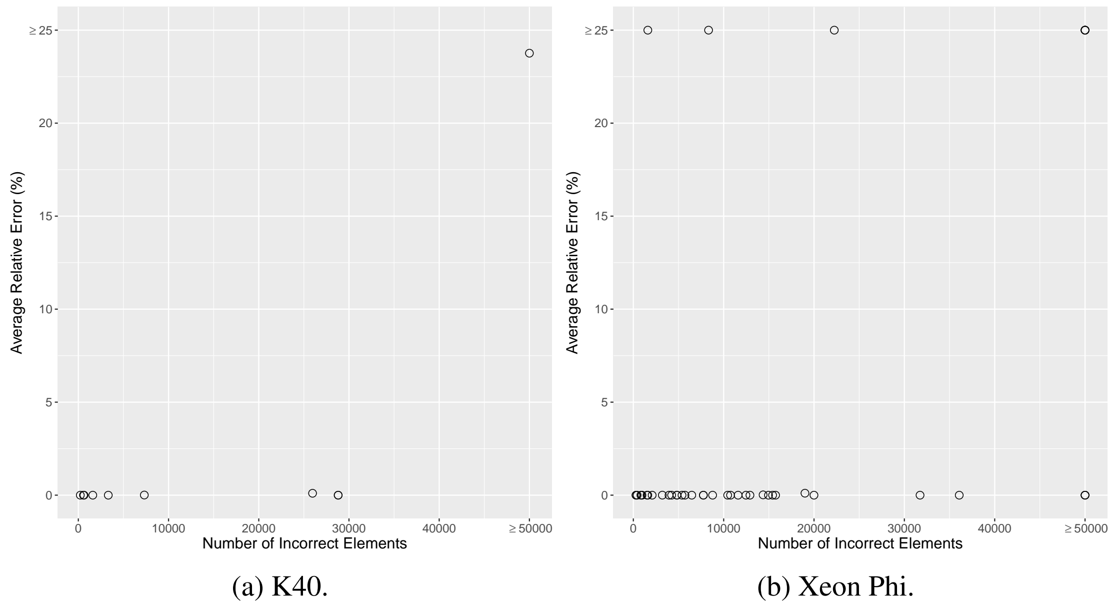
Source: The Author

5.2.2 *LavaMD*

Mean relative error and number of incorrect elements for *LavaMD* are reported in Figures 5.4a and 5.4b. As the mean relative error is extremely high for *LavaMD* we represent errors with a mean average error up to 20,000%. Executions with a mean average error higher or equal to 20,000% are shown as 20,000% to improve figure quality). We hypothesize that the higher relative error of *LavaMD* compared to *DGEMM* is related to the exponentiation operation used when computing particle interactions, which can turn small value variations into large differences. The number of incorrect elements, on the contrary, is low and concentrated for the K40. Xeon Phi shows a higher number of corrupted elements than the K40 but a much lower average error. Although K40 simulates more particles than Xeon Phi (192 and 100 per box, respectively), Xeon Phi is affected by a larger number of incorrect elements. However, those corrupted elements for the Xeon Phi have an overall lower difference with the expected values.

Spatial locality and relative error for *LavaMD* is presented in Figures 5.5a and 5.5b. K40 has no errors with a relative error lower than 2% while Xeon Phi has only about one tenth of errors lower than the 2% threshold. Spatial locality highlights that most of the errors for Xeon Phi are cubic and square. K40 corrupted output affected by cubic and square error patterns are 40% to 60% of the total. The spatial locality for Xeon Phi is related to the larger number of incorrect elements which corresponds to an increased spatial

Figure 5.6: *HotSpot* Mean relative error and Incorrect Elements.



(a) K40. (b) Xeon Phi.

Source: The Author

area of corrupted data. As Xeon Phis have larger shared cache memories, it is easier for a single impinging particle to affect data used by multiple cores when running *LavaMD*.

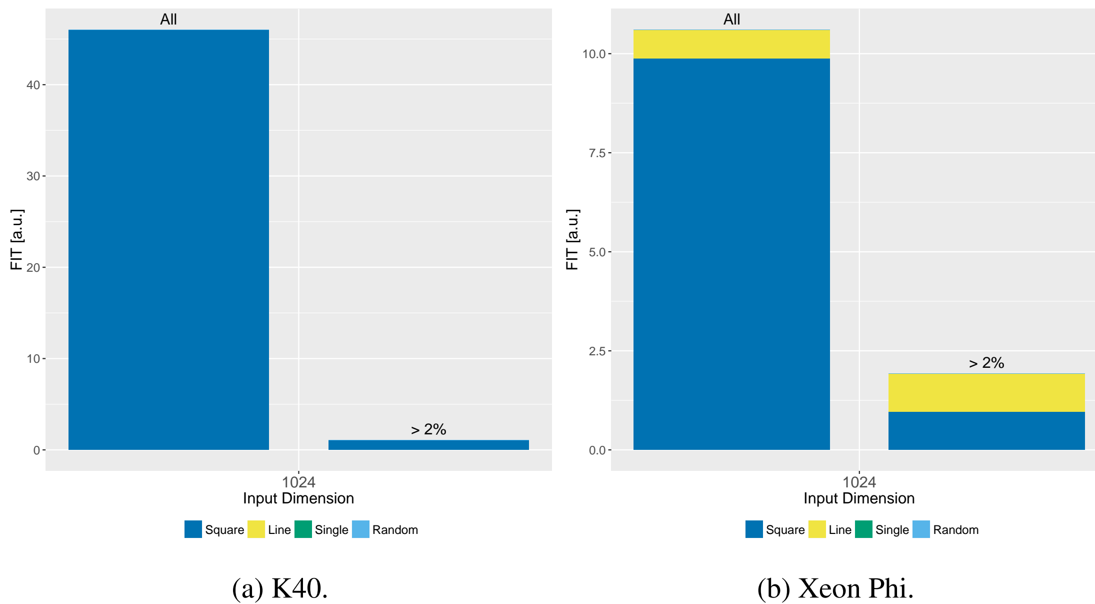
The percentage of K40 corrupted outputs with cubic and square error patterns are decreasing as the input dimension grows (55% of all corrupted output for 15 cubes, 50% for 19, and 42% for 23). With a larger input, more threads have to be scheduled and more data has to be read and written. The increased pressure in the GPU reduces the sharing of resources like caches, increasing the isolation between blocks of threads. This isolation, in turn, reduces the probability of corrupted data to be shared among many blocks, causing less cubic and square errors.

For the K40, *LavaMD*'s FIT rate increase with input size is only about 30% from one input size to the next one, definitely less than for *DGEMM*. This is only in apparent contrast with (1) and (2). In fact, *LavaMD* makes heavy usage of local memory (≈14 KB per block of threads), which limits the number of active threads at any given time on the K40. Thus, the increase in the number of active threads is limited for *LavaMD*, reducing the impact of the scheduler strain.

5.2.3 *HotSpot*

HotSpot values for mean relative error and incorrect elements are shown in Figures 5.6a and 5.6b. *HotSpot* shows an extremely low mean relative error (lower than 25%

Figure 5.7: *HotSpot* spatial locality and magnitude.



(a) K40.

(b) Xeon Phi.

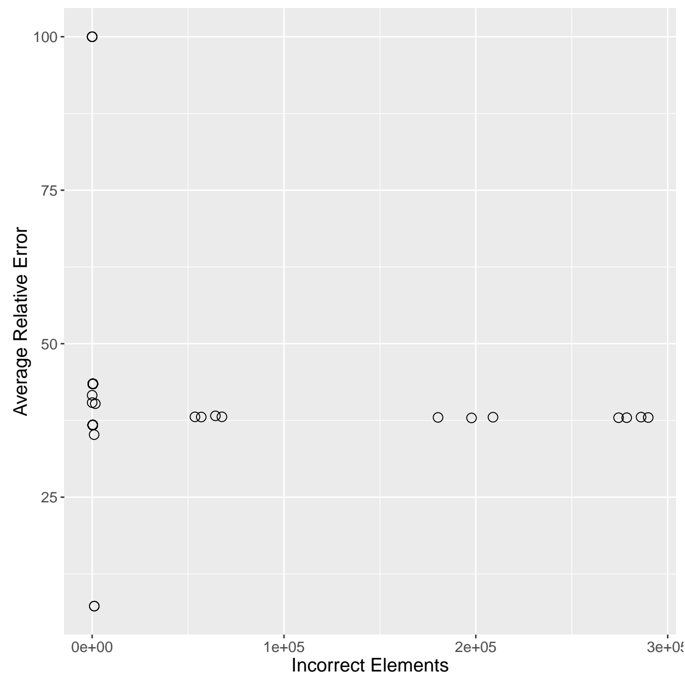
Source: The Author

in all cases) independent of the number of incorrect elements for both architectures, which is due to intrinsic algorithm characteristics. *HotSpot* simulates energy dissipation taking into consideration the power input and the temperature of nearby cells. Therefore, errors will eventually dissipate as the result tend to reach an equilibrium. Analyzing the number of incorrect elements in Figures 5.6a and 5.6b, it is clear that Xeon Phi shows a greater tendency to have multiple errors than K40. K40, in fact, has at most about 50,000 incorrect elements in the output while Xeon Phi experienced up to 130,000 incorrect elements in the output (executions with a number of incorrect elements higher or equal to 50,000 are shown as 50,000 to improve figure quality).

Figures 5.7a and 5.7b depict the spatial locality and relative errors for *HotSpot*. Both architectures presented only square and line errors. The computation of each cell takes as direct input the values of the neighbor cells. Therefore, one single error will affect neighbor cells in the next iteration, always increasing spatial locality criticality. Considering only errors above 2%, *HotSpot* shows the most expressive results as we could consider as correct about 80% to 95% of faulty executions for Xeon Phi and K40, respectively.

HotSpot can greatly recover from errors naturally due to algorithm characteristics. Most of the faulty executions presented errors smaller than 2%. *HotSpot* is intrinsically robust and considering all mismatches as an error would erroneously decrease its resilience. Therefore, one can imprecisely classify *HotSpot* with a radiation sensitivity up to 95% higher considering any mismatch with the expected value as the sole metric.

Figure 5.8: *CLAMR* Mean relative error and Incorrect Elements for Xeon Phi.



Source: The Author

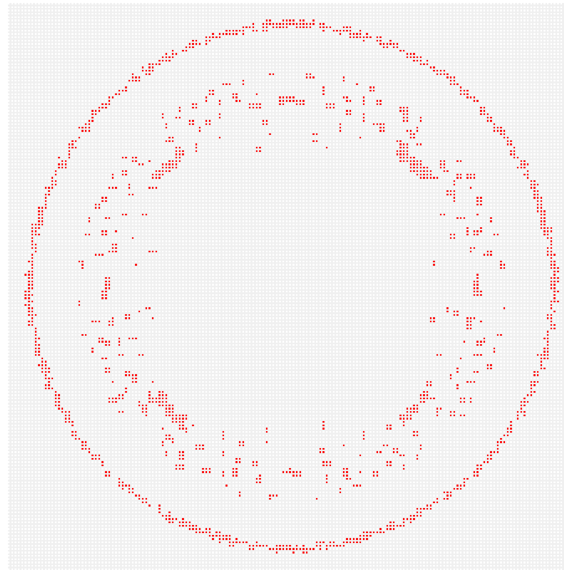
The evaluation of neighbors to detect disparities coming from errors in Stencil-like applications like *HotSpot* can be difficult. The criticality results show that an error could be dissipated to neighbors leading to small disparities but with a significant accumulated error in the affected elements. Thus, to detect an error, the checking routine would need to be executed constantly, reducing performance. The system entropy could be evaluated to detect a widespread error in stencil-like applications, especially if the system is isolated and the entropy needs to be constant. However, for non-isolated systems, if the growing or lowering of system entropy is well-behaved, the entropy could be checked at regular time intervals to detect disturbances caused by induced errors. The time interval could be adjusted to better detect errors without affecting performance too much.

5.2.4 *CLAMR*

Fig. 5.8 shows the mean relative error and number of incorrect elements for *CLAMR* on Xeon Phi. We do not have the results for K40 as *CLAMR* is a LANL's proprietary workload to be used in supercomputers like Trinity, which will be based on Xeon Phis.

CLAMR shows a mean relative error between 25% and 50% while incorrect elements are definitely high. When mapping the incorrect elements to the 2D grid, most of the forms were similar to the one presented in the Fig. 5.9. We can see that a wave

Figure 5.9: *CLAMR* Error Locality Map. The output result is represented as a 2D matrix. Red dots are incorrect elements.



Source: The Author

of incorrect elements was propagating confirming the fault injection analysis performed in (GUAN et al., 2015). Incorrect elements are, then, not isolated, affecting first the neighborhood and propagating as a wave, increasing the number of incorrect elements as the executions continue.

All the faulty elements of *CLAMR* have relative errors greater than 2%. Square errors amount to 99% of spatial locality as the errors propagate to all directions as depicted in Fig. 5.9. *CLAMR* works with the shallow waters equations considering momentum as well as mass. *CLAMR* errors may change the total mass of the system and will not be recovered as the execution continue, on the contrary, because of the mass conservation principle, the error will keep affecting the solution. Therefore, among all the codes studied in this work, the error criticality of *CLAMR* was the most sensitive to radiation-induced errors.

The sparse spatial location of incorrect elements with a moderate to low relative error makes it hard to provide efficient techniques for detecting errors. Similar to *HotSpot*, one of the few ways that can be used to detect errors is to evaluate the whole system taking advantage of the mass conservation principle, where the summation of all the incorrect elements' errors can lead to a detectable mass difference. This mass check technique has already been implemented in *CLAMR* and fault injection showed a fault coverage of 82% (ATKINSON et al., 2014). Furthermore, the load imbalance of the algorithm can provide opportunities to include mass-consistency checking routines that introduce little overhead to the overall execution time.

5.3 Discussion

The tested codes were selected as they stimulate specific resources, and have peculiar control flow or arithmetic characteristics. Thus, it is reasonable to correlate the particular behaviors observed for Xeon Phi and K40 with the code proprieties. Additionally, in some cases we can extend the experimental criticality analysis to other codes that share the same principles and data structures of the tested codes.

DGEMM is part of several applications and similar codes are even used as the standard HPC performance benchmark. Our analysis shows that K40 provides a lower error criticality (i.e., smaller difference with the expected value and fewer corrupted elements) for applications using *DGEMM*. We believe this is an intrinsic characteristic of GPUs, which has shortened and faster pipelines compared to CPUs. As a result, purely arithmetic operations, that are not based on control-flow instructions, are likely to be executed in a faster and more reliable way on a GPU.

Solvers using FDM like *LavaMD* will have a lower relative error on Xeon Phi, in contrast to *DGEMM*. Transcendental functions play a key role in FDMs performance and reliability. In the case of *LavaMD*, the exponentiation operations can turn small value variations into large differences. This is especially critical for the K40, for which all the SDCs are significantly different from the expected value. A hypothesis is that the transcendental function unit in the K40 is more prone to corruption. Its reliability should be improved in the future. Also, *LavaMD* performs dot products between particles which prevent the attenuation of transient errors with other correctly calculated particles.

While the Xeon Phi seems more resilient than the K40 when executing *LavaMD*, it produced more incorrect elements, leading to a more widespread spatial locality. To choose the platform in which to execute an FDM algorithm, it is fundamental to evaluate the trade-off between having more incorrect elements with lower relative errors (so the Xeon Phi) or the contrary (so the K40). Such a trade-off strictly depends on how FDMs outputs are used.

Stencil applications have been proved to be the most resilient ones. *HotSpot* showed that most of the errors have less than 2% of relative error. K40 seems slightly more resilient than Xeon Phi as the former shows less incorrect elements than the latter. We believe this behavior to be common for stencil applications that iteratively update the solution based on the current state. For these applications a transient fault could modify the current simulation state but, in the following iterations, the error is smoothed and

filtered.

Fluid dynamics like *CLAMR* are less reliable, especially simulations that involve invariants such as mass or energy conservation. The impact of errors in such algorithms only increases with execution time as the invariant is now altered, affecting neighbor elements in the following iterations.

Based on our experimental analysis we can compare the reliability of some peculiarities of Xeon Phi and K40 architectures. Xeon Phi shows a tendency to have more incorrect elements than K40. Xeon Phi has larger caches than K40, so its data is not evicted as often. Hence, corrupted data, once in the caches, will be used by more elements before eviction. As a result, the same error spreads affecting several output elements.

The comparison of results with different input sizes for *DGEMM* and *LavaMD* highlights that hardware scheduler makes the FIT rate dependent on the number of instantiated threads. On the contrary, the Xeon Phi operating system seems less prone to be corrupted. It is worth noting that the hardware scheduler may be more efficient, reducing the execution time and, thus, the number of neutrons that hits the device during computation. Other architectural decisions like long pipelines or large caches that keep alive data for a long time during computation modify both the code reliability and execution time, enhancing performance but leaving data more exposed to radiation strikes. Thus, the architectural design must tune the performance gain obtained by such decisions with the reliability issues incurred (REIS et al., 2005a).

Finally, the spatial locality and magnitude of the errors measured for the different applications and devices can help users understand incorrect results generated from radiation-induced errors, and guide the usage of detectors and replication mechanisms (BERROCAL et al., 2016).

6 FAULT INJECTION CRITICALITY ASSESSMENT

In this chapter, we aim to improve the understanding of the resilience problem for HPC application. We will perform the criticality assessment for results from software fault injection campaign. We use fault injection to obtain extra information that cannot be obtained by radiation experiments.

6.1 Methodology

In this section, we describe the fault injection tool used to perform the criticality assessment. Then, we detail the device under test, and the selected algorithms chose to help understand the reliability issue.

6.1.1 CAROL-FI Fault Injector

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. The goal, in fact, is not to measure the sensitivity to a transient fault of an application, as we gathered this information with the neutron beam experiments performed in previous sections, 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. Therefore, we implemented a fault injector called CAROL-FI (available at (OLIVEIRA, 2017)).

CAROL-FI allows the injection of various fault models and correlates 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 gathering 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.

The fault injector designed for this analysis 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 first 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 6.1.1.1. 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 $4\times$ the normal execution time, with a worst case of $8\times$, 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 with the GDB.

6.1.1.1 Fault Models

The fault injection 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 a low level to code level. The four models are:

- **Single**: flip a single random bit
- **Double**: flip two random bits from the same variable
- **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 3.1.2. 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 (FANG; OATES, 2016). SECDED ECC normally triggers application crash when a double bit error is detected. The 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 5.2, the Xeon Phi error rate is comparable to NVIDIA GPUs, 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.

6.1.2 Device Under Test

The CAROL-FI Fault Injector was first developed using the Intel Xeon Phi as the platform. Thus, we use the Xeon Phi, described in section 5.1.2, as the device under test. However, CAROL-FI has been extended to NVIDIA GPUs as well.

6.1.3 Selected Algorithms

We selected the algorithms *CLAMR*, *DGEMM*, *HotSpot*, and *LavaMD* detailed in sections 4.1.3 and 5.1.3. We have also selected *LUD* detailed next.

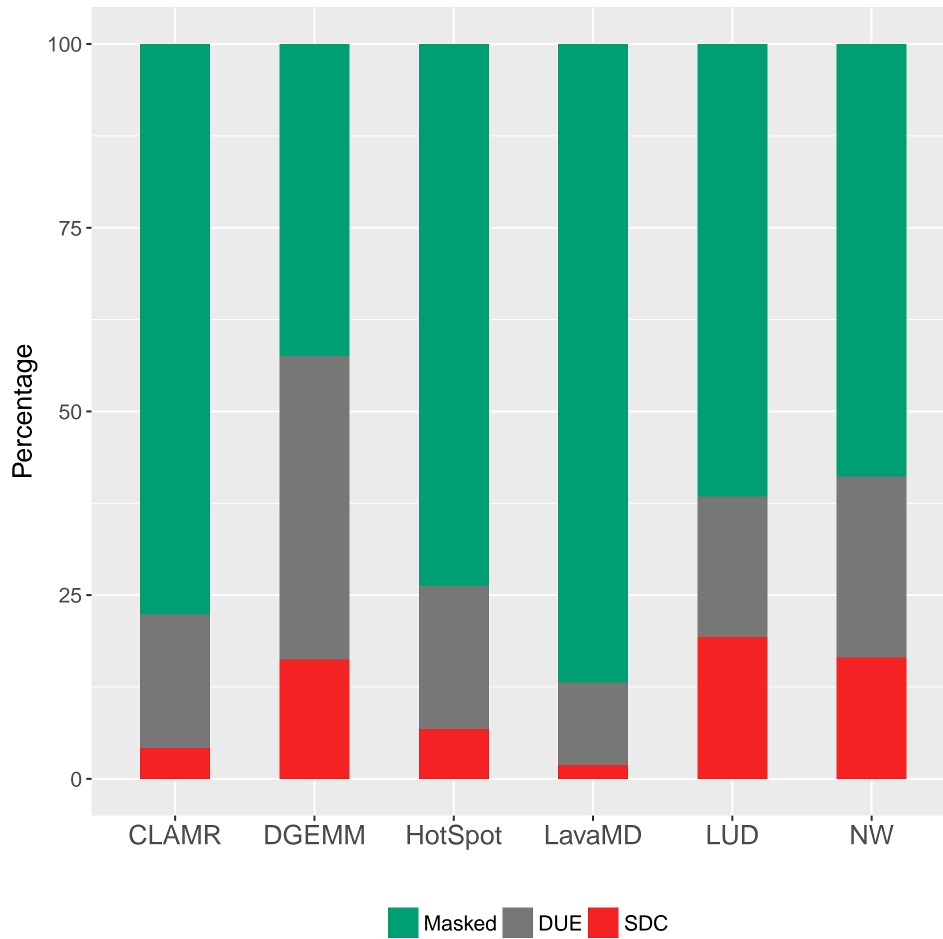
LUD is also a dense linear algebra as *DGEMM*. However, *LUD* uses less memory than DGEMM and has more interdependencies resulting in an algorithm that is less compute-bound than *DGEMM*. *LUD* decomposes the input matrix into a product of lower and upper triangular matrices (CHE et al., 2009). *LUD* uses a static partitioning of the data and has a regular memory access pattern.

6.2 Results

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 6.1 presents the percentage of faults that are masked or cause an SDC or DUE for each benchmark presented in sections 4.1.3 and 5.1.3. 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 4.2.2, it is not possible to directly correlate our beam experiments with CAROL-FI results. Figure 6.1 shows the probability of corrupted portions of the source code to affect the execution.

Figures 6.2 and 6.3 show the Program Vulnerability Factor (PVF) for SDC and DUE, respectively, for each fault model described in Section 6.1.1.1. The different fault

Figure 6.1: Outcomes of fault injections.

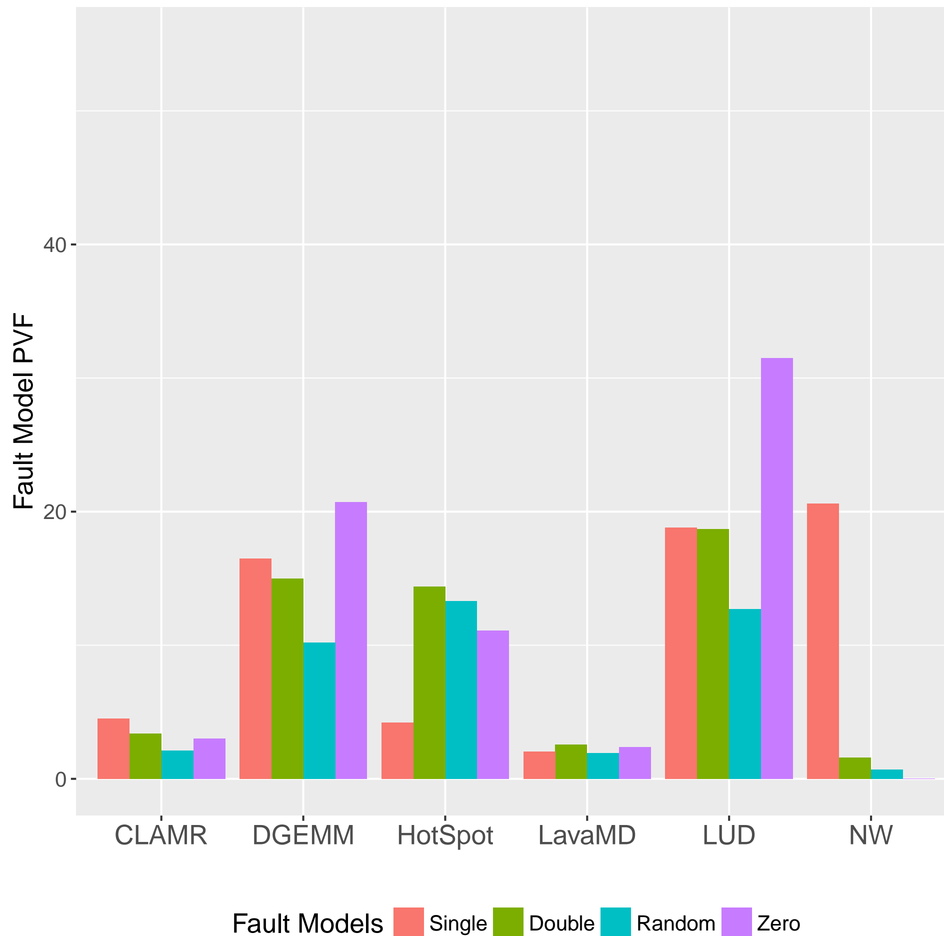


Source: The Author

models yield quite different PVFs 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 6.4 and 6.5, respectively). Please note that Figures 6.4 and 6.5 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

Figure 6.2: SDC's PVF of the benchmarks for the different fault models.



Source: The Author

100%.

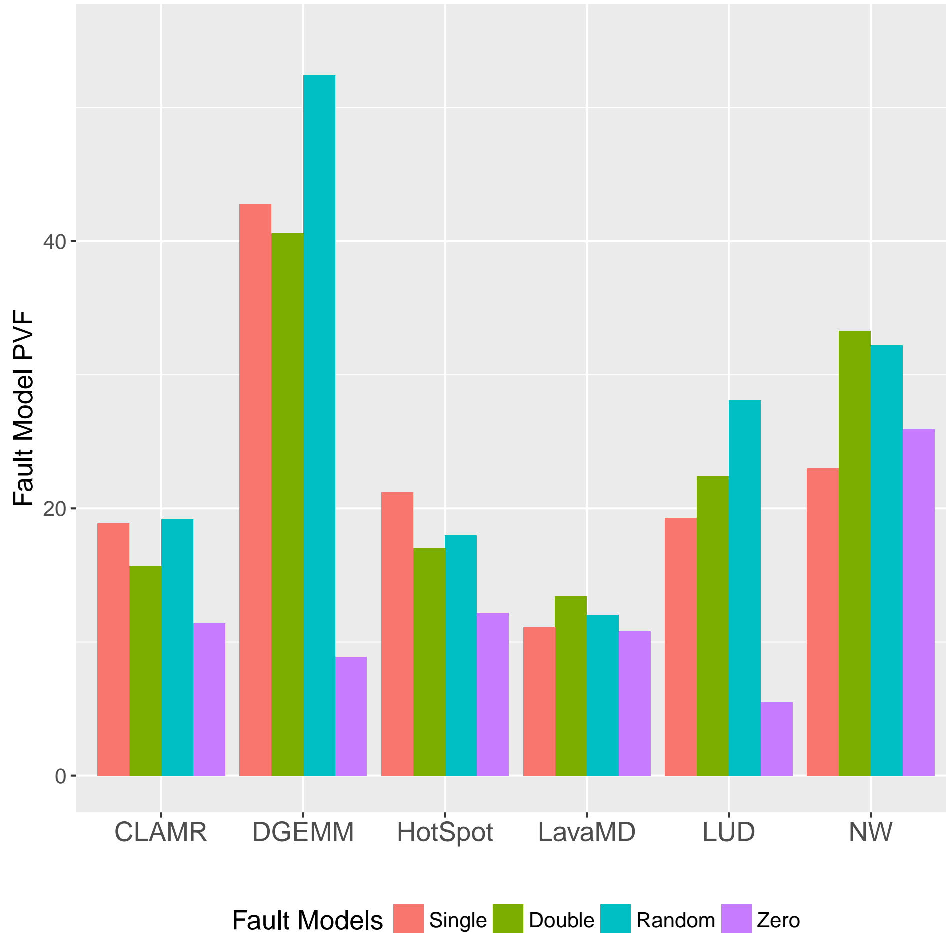
In the following, we analyze each benchmark individually to demonstrate how CAROL-FI can be used to derive information about benchmark sensitivity and also provide insights on how to improve the resilience of each benchmark.

DGEMM

As shown in Figure 6.1, 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 their own copy of the loop control variables, increasing the memory portion used to store

Figure 6.3: DUE's PVF of the benchmarks for the different fault models.



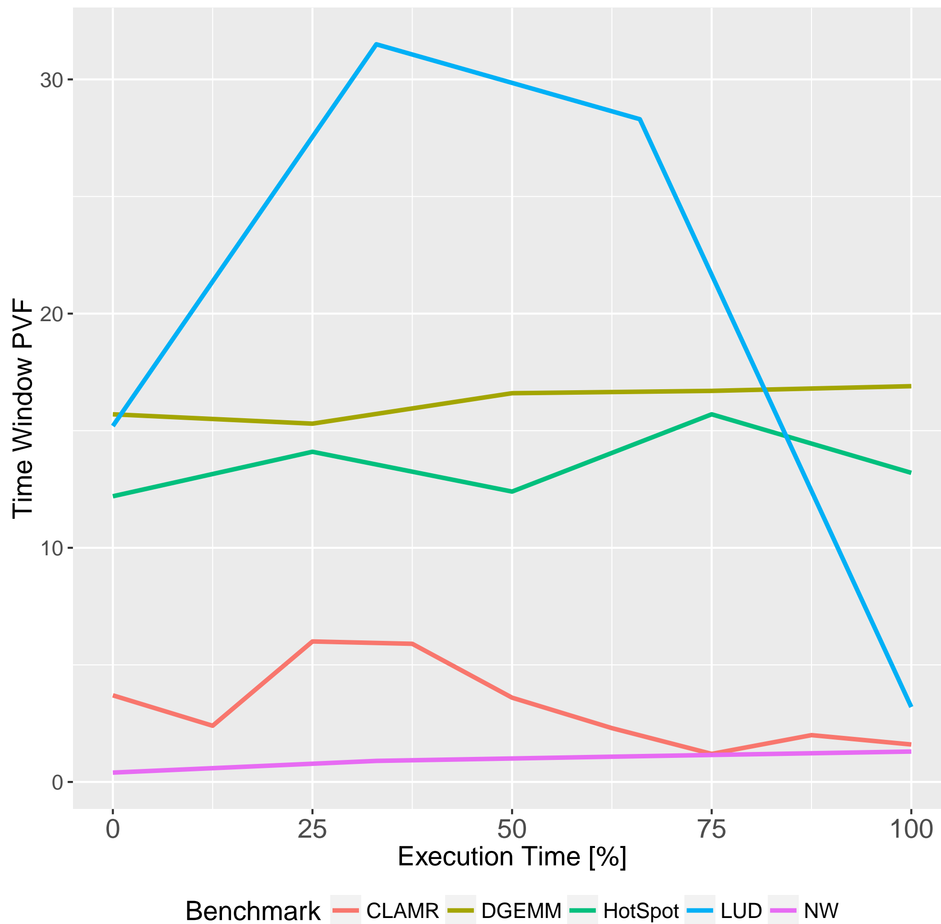
Source: The Author

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 6.2 and 6.3 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 6.3, we find that Random and Zero have opposite behaviors. Random and Zero models have a higher likelihood to 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 6.4 shows

Figure 6.4: The dependence of the SDC's PVF of the benchmarks on the execution time window.



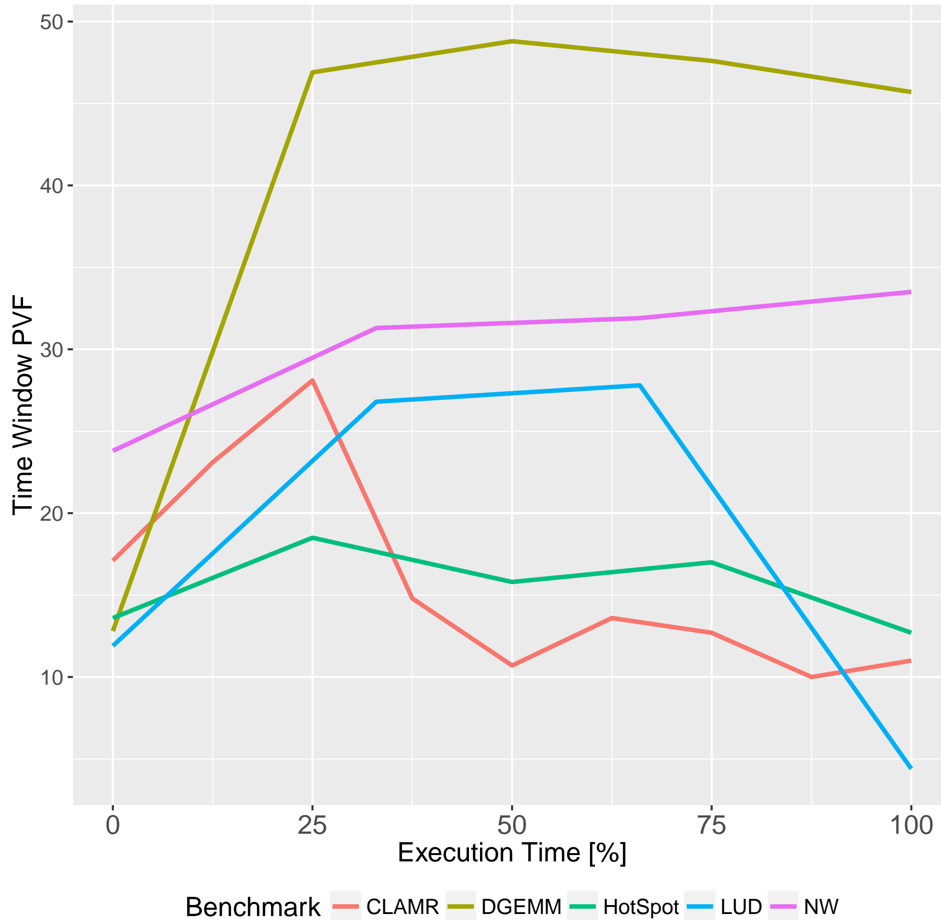
Source: The Author

that the SDC error rate remains unchanged between time windows. However, in Figure 6.5 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 on 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 structures 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.

Figure 6.5: The dependence of the DUE's PVF of the benchmarks on the execution time window.



Source: The Author

CLAMR

Injected faults are masked 75% of the time in *CLAMR*, as shown in Figure 6.1. 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 6.2). For DUEs, only the Zero model yields a different rate than the other models, as can be seen in Figure 6.3. 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 Figures 6.4 and 6.5 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 (GUAN et al., 2015). *CLAMR* becomes more sensitive when the number of active cells reaches its maximum value, which can be automatically set by the algorithm itself.

The 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 (ARGYRIDES et al., 2009) 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 a fault in Sort or 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 6.1. Most of the observed SDCs and DUEs are caused by injections in **constant and control variables** used during computation. The 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 6.2 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 6.4 and 6.5.

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 iteratively, and thus, errors in intermediate values will have a 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 6.1 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 6.2 and 6.3 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 6.1).

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 6.2 and 6.3 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 6.1 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 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 6.2 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 6.3), 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.

6.3 Discussion

As we can see from radiation experiment results in previous chapters, 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 5.7) so a simple mitigation technique can provide the desired level of resilience.

7 RELIABILITY AND CRITICALITY COMPARISON

This chapter performs a thorough comparison of reliability and criticality of several HPC applications in different architectures. Such comparison aims to better understand the code-dependent and architecture-dependent aspects of the resilience problem. Thus, we used the methodologies presented in previous sections, and we did not limit the evaluation to HPC devices alone. We use a representative set of algorithms and a broad range of devices, from HPC to embedded ones.

7.1 Methodology

In this section, we list and detail the devices tested in this work. Then, we described the algorithms used to perform the comparison. Finally, we present the methodology for neutron bean and fault injection experiments.

7.1.1 Devices Under Test

In this study, we consider NVIDIA GPUs (Kepler K40, Tegra X1, and Titan X), Intel Xeon Phi, AMD Kaveri APU, and ARM Cortex-A9. The NVIDIA K40 (**Kepler**) and Intel Xeon Phi (**KNC**) are detailed in section 5.1.2, the remaining devices are detailed below.

The ARM Cortex-A9 (**ARM**) is embedded in a Zynq-7000 APSoC (XILINX, 2016), designed by Xilinx. We use one core of the Zynq board dual-core ARM Cortex-A9 processor built in a $45nm$ CMOS TSMC technology and containing about 26 million transistors. The processor has two 4-way set-associative 32KB L1 caches (data and instruction) per core, and one 8-way set-associative L2 cache with 512KB shared between both cores. A dual-ported 256KB on-chip SRAM memory (OCM) is shared between the processor and the FPGA.

The Tegra X1 (**Maxwell**) is the embedded version of the $Maxwell$ GPU fabricated in $20nm$ standard TSMC CMOS technology. It includes an ARM A57 quad-core CPU and a GPU with 256 CUDA cores divided into two Streaming Multiprocessors (SMs). Each GPU SM has 64KB of L1 cache and 32KB of registers file capacity, and 256KB of L2 cache shared between SM's cores. The X1 operates at 1GHz. In this work, the ARM

Table 7.1: Devices under test specifications summary.

Architecture	**Steamroller**	**KNC**	**Kepler**	**Pascal**	**Maxwell**	**ARM**
Tested Processor	GPU	Accelerator	GPU	GPU	GPU	CPU
Technology	TSMC 28nm	Intel 3-D TriGate 22nm	TSMC 28nm	TSMC 16nm FinFET	TSMC 20nm	TSMC 45nm
L1 Cache [KB]	256	3648	960	1344	128	128
L2 Cache [KB]	4096	29184	1536	3072	256	512
# Cores	512	57	2880	3584	256	2
Core Frequency [MHz]	720	1100	745	1400	1000	667
# Transistors [B]	2.41	N/A	7.10	12	N/A	0.026

A57 is used just as a host for the GPU.

AMD Kaveri APU (**Steamroller**) is the embedded A10-7850K built with the $Steamroller$ architecture with a $28nm$ standard CMOS technology. The GPU is an AMD Radeon R7 Series containing 512 cores with 720MHz each. The A10-7850K has two sets of 96KB 3-way set associative shared L1 instruction caches, and four sets of 16KB 4-way set associative L1 data caches. Also, the APU has two sets of 2MB 16-way set associative shared L2 caches.

The NVIDIA Titan X (**Pascal**) is designed with the $Pascal$ architecture, in $16nm$ TSMC FinFET technology. Titan X has 3584 CUDA cores split across 28 SMs, each core running at a 1.4GHz base clock. NVIDIA Titan X has 12GB of GDDR5X SDRAM memory. Each SM shares a 256KB register file and 48KB L1 cache. All the SMs share 3MB of L2 cache (NVIDIA, 2016).

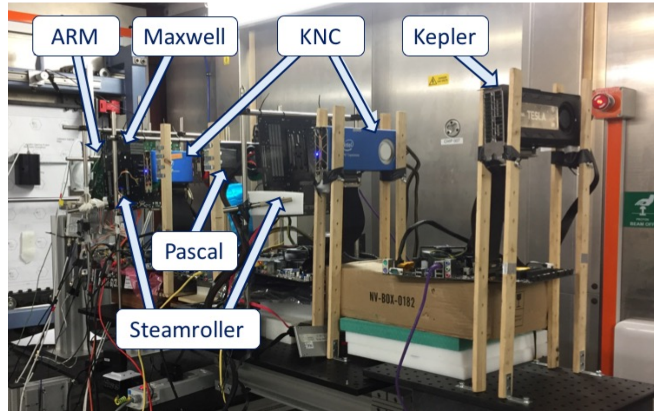
A summary of the tested device's specifications can be found in Table 7.1. It is worth noting that the different transistor layouts and the different amount of available resources will impact the probability of a particle generating bit-flips or logic errors (BAUMANN, 2005; NOH et al., 2015) while microarchitectural differences will affect the way low-level faults propagate to a visible program output. Section 7.2 of this study evaluates both aspects.

7.1.2 Selected Algorithms

We tested algorithms detailed in sections 4.1.3 and 5.1.3. However, we also included sorting algorithms detailed below.

Quick sort is a traditional sorting algorithm. The sorting problem is solved by a recursive procedure which is divided into three phases. The first phase chooses a $pivot$ element on the array. The second phase orders all the input elements relative to the chosen $pivot$. The third phase divides the input elements into two parts centered on the chosen $pivot$, then recursively calls the same procedure to each of the subsequently divided arrays

Figure 7.1: Radiation test setup at ChipIR. Neutrons come from the left-side of the picture.



Source: The Author

of elements.

Merge sort is an optimized sorting algorithm. The Merge sort algorithm follows the *divide and conquer* strategy, which consists in dividing a given problem into several smaller problems. Smaller problems are then solved by simpler methods separately. Merge sort accelerates the sort problem by solving several smaller problems concurrently on the multiple cores available. This *divide and conquer* would make Merge sort too inefficient for ARM processor, so we tested it only on parallel devices.

The input size defines the size of the problem to be solved. We tailor input sizes to fully occupy the available resources of each device. A not fully used device would, in fact, result in a lower FIT because of unused area. Table 7.2 lists the number of output elements for all the codes. Input values were randomly generated, carefully balancing the number of 1s and 0s and selecting values that could not result in overflow during computation.

7.1.3 Neutron Beam Test Experimental Setup

The data we present has been gathered from several radiation experiments performed at the ChipIR facility of the Rutherford Appleton Laboratory (RAL) in Didcot, UK, and the at Los Alamos Neutron Science Center (LANSCE) facility of the Los Alamos National Laboratory (LANL). We test each device and configuration at both ChipIR and LANSCE. FIT rates were similar (within a margin of error) across facilities.

Figure 7.1 shows part of our setup at ChipIR. A total of 2 ARM, 2 Steamroller APUs, 6 Maxwells, 3 Kepler, 1 Pascal, and 3 KNC were irradiated. When measuring FIT rates, we consider both board position and the number of boards between the neutron

Table 7.2: Problem size, execution time, and FIT rates for the tested codes.

Benchmark	FIT [a.u.]		Execution Time [s]	# Output Elements [M]	FIT [a.u.]		Execution Time [s]	# Output Elements [M]	FIT [a.u.]		Execution Time [s]	# Output Elements [M]
	SDC	DUE			SDC	DUE			SDC	DUE		
	Kepler				**Kepler-ECC**				**KNC**			
GEMM	16.267	1.656	0.017	4	1.659	2.485	0.0172	4	14.186	2.471	0.19	4
LavaMD	163.969	19.769	0.059	∼0.183	11.891	46.450	0.059	∼0.183	11.255	24.774	0.72	∼0.654
Hotspot	21.676	18.450	0.2	1	3.087	17.630	0.233	1	22.904	24.774	0.37	1
NW	20.016	28.690	0.077	256	6.773	90.558	0.086	256	22.182	25.879	0.12	64
Quick sort	105.807	36.775	2.265	64	1.165	20.535	2.322	64	4.366	5.867	7.78	64
Merge sort	159.401	43.301	0.799	64	20.860	31.155	0.804	64	5.102	5.639	8.21	64
	Maxwell				**Pascal**				**Steamroller**			
GEMM	20.727	12.802	0.165	1	11.5	1	0.053	4	10.831	4.829	1.5	1
LavaMD	37.793	12.311	0.684	∼0.039	29.165	3.170	0.144	∼0.183	19.906	2.072	1.194	∼0.039
Hotspot	19.532	11.936	3.919	1	9.035	1.348	0.131	1	9.451	6.037	1.898	1
	ARM				**Kepler**				—			
MxM	6.236	1.751	12.437	∼0.238	25.878	8.617	0.48	4	—	—	—	—
FFT	19.357	3.925	5.745	∼0.031	196.126	72.436	0.4	16	—	—	—	—
Quick sort	18.652	4.666	1.430	∼ 0.047	105.807	36.775	2.265	64	—	—	—	—

source and the device under test. The methodology for beam experiments is detailed in section 4.1.

7.1.4 Fault Injection Frameworks

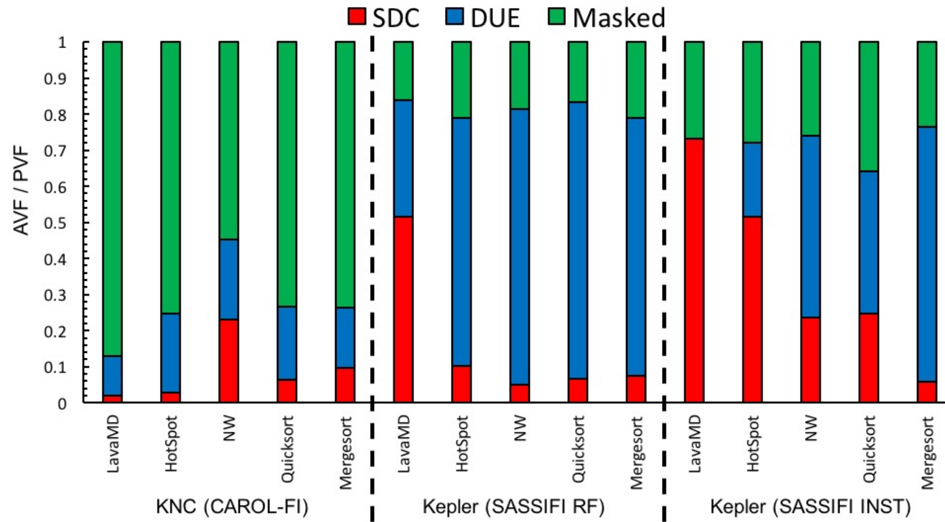
For our analysis, we inject errors on Kepler architecture using **SASSIFI** and on KNC using **CAROL-FI** detailed in section 6.1.1.

SASSIFI is a tool used to inject faults into NVIDIA GPUs. It injects transient errors in the GPU's ISA visible state, including general purpose registers, memory values, predicate registers, and condition registers (HARI et al., 2017). Prior work used GPUQin (FANG et al., 2014) to evaluate CUDA benchmarks reliability. We choose SASSIFI mainly because: (1) it is much faster than GPUQin, (2) it injects faults both in register file and instructions. We inject faults both in the instruction output (INST) and in the register file (RF). INST injections measure the PVF, while RF injections measure the register file AVF. We use two fault injection models: inserting single or double bit-flips and random values (HARI et al., 2017). The types of faults can be injected into a single thread or every thread in a warp.

7.2 Reliability Evaluation

In this section, we discuss and correlate fault-injection results and beam data. Then, we analyze the Mean Workload Between Failures to consider the performance-reliability trade-off. Based on architectures and algorithms characteristics we can identify

Figure 7.2: Fault injections results on the KNC and Kepler. Injections are at source code level for the KNC and both at register file (RF) and at the output of instructions (INST) for the Kepler.



Source: The Author

some generic device-dependent and code-dependent reliability behaviors.

GEMM, LavaMD, NW, and Hotspot are specifically tailored for parallel devices and would be very inefficient (and not representative) when executed on the ARM. On the ARM we execute MxM, FFT, and Quick sort. To have a baseline comparison between ARM and HPC devices we execute MxM and FFT also on the Kepler (but not on the other HPC devices, for lack of beam time). We collected more than 100 SDCs and 100 DUEs per reported code, and we injected more than 10,000 faults per configuration, to have Normal's 95% confidence intervals lower than 10% of the presented values.

7.2.1 Fault Injection Results

Figure 7.2 shows the fault injection results for LavaMD, Hotspot, NW, Quick and Merge sort as executed on the KNC and Kepler. It was not possible to inject errors in GEMM, as it is a proprietary code.

Unfortunately, a unified fault injector for all the devices we are considering does not exist. However, while a direct comparison between the fault-injectors is not possible, we can use our results to compare the errors propagation probability between devices executing the same code.

Comparing KNC and Kepler data in Figure 7.2, we can conclude that the probability of errors to propagate depends both on the code and on the architecture. In particular,

for the KNC, NW is the code with higher PVF for SDCs, while for the Kepler, LavaMD is the most vulnerable code (for both RF and INST injections). NW adapts better to GPUs architectures because it is very suitable to be parallelized, while LavaMD extensively uses transcendental functions, which are inefficient (and less reliable) on GPUs. We anticipate that beam data confirms this result.

Additionally, the DUE AVF/PVF is almost independent on the code for the KNC (variations are lower than 20%) while for the Kepler the probability for injections to generate a DUE strongly depends on the algorithm. This behavior is justified by the fact that codes run on the top of an operating system on the KNC, while they run bare-to-the-metal on the Kepler. The presence of an operating system significantly influences the DUE rate (SANTINI et al., 2016).

The comparison between RF and INST injections on the Kepler can also be used to predict the effectiveness of ECC. Errors injected in the RF are likely to be masked by ECC. Additionally, as data in the cache needs to be loaded to registers to be digested, RF AVF is a reasonable estimation of how cache errors (protected by ECC) would affect computation. From Figure 7.2 we can conclude that even ECC effectiveness is code-dependent. ECC is likely to reduce the SDC rate for LavaMD and Hotspot significantly. Additionally, ECC is likely to reduce DUE induced by data corruption (errors in indexes, control variables, etc.), mostly for Quick and Merge sort.

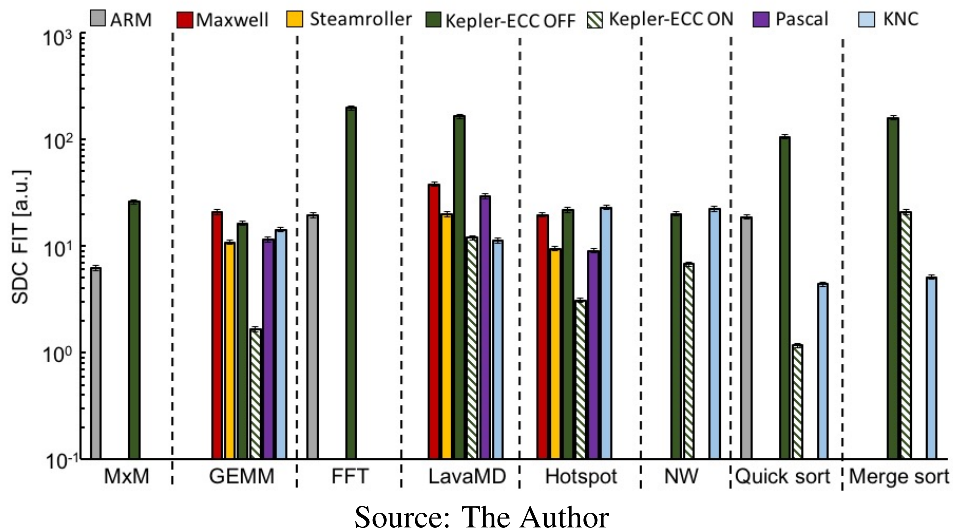
7.2.2 Beam Experiments Results

Figures 7.3 and 7.4 show the results of the neutron beam experiments as normalized Failure in Time (FIT) rate for the tested architectures. Reported data have been normalized to the lowest FIT rate (GEMM DUE as executed on the Pascal) to prevent the leakage of business-sensitive data while allowing a direct comparison between devices and codes error rates. Experimental data is divided into SDC (Figure 7.3) and DUE (Figure 7.4).

In this section, we quantify the SDC FIT counting as faulty all executions with any bit mismatch between the output of the program and the expected, error-free, output. In Section 7.3 we qualify the observed SDCs also evaluating the number of corrupted elements and the differences between the corrupted and expected data.

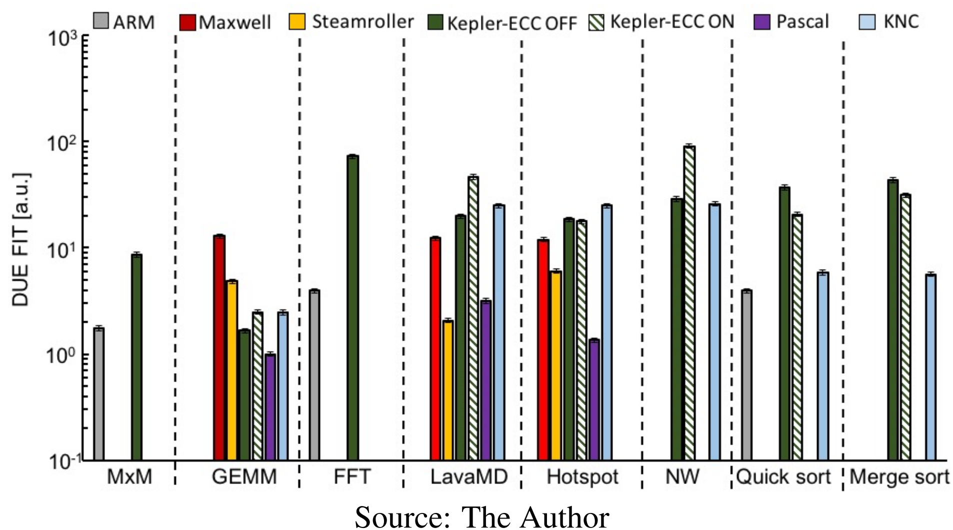
As expected, the FIT rates vary significantly between devices and codes. The SDC FIT rate for the same code executed in two different devices can vary of up to 1 order of

Figure 7.3: Beam experiment results, organized as relative FIT rate for SDC.



Source: The Author

Figure 7.4: Beam experiment results, organized as relative FIT rate for DUE



Source: The Author

magnitude (as the cases of LavaMD or Quick sort). Figure 7.3 shows that the SDC rate of Kepler with ECC disabled is much higher than other devices for MxM, FFT, LavaMD and, mostly, Quick and Merge sort. These are the algorithms that use more memory and suffer from longer memory latencies. Devices with a higher amount of unprotected memory (caches and registers) will have more data exposed while waiting to be digested, resulting in a higher chance of experiencing a radiation-induced fault. GEMM, Hotspot, and NW are compute-intense algorithms, designed to lower memory transfers and to digest data as soon as possible. GPUs are particularly indicated for compute-intense codes, as they can take advantage of the higher number of computing units to minimize latencies. On GPUs, the exposure time is then significantly reduced. For the KNC and ARM, on the contrary,

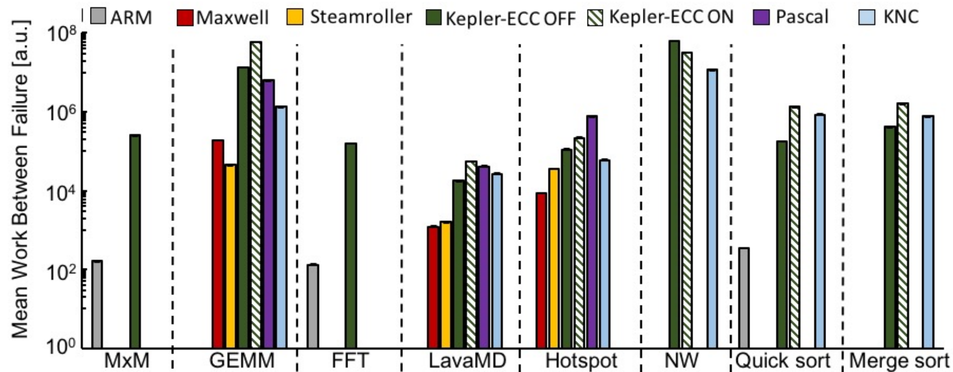
it is not possible to exploit the algorithm compute-intense structures completely. In fact, KNC and ARM SDC rate are not significantly reduced when executing compute-bound codes. The KNC shows a particularly low SDC FIT rate for sorting which, as discussed in the following, is granted by ECC protection.

The DUE FIT rate has a significant device-dependent component (probability of corruption of the scheduler, host-device interface, operating system, when available) and a code-dependent component (number of control-flow operations, data-based branches, etc.). The comparison between Figure 7.3 and Figure 7.4 shows that the DUE rate is much less code-dependent than SDC rate. The device DUE rates vary at most of 30% between MxM, LavaMD, Hotspot, and NW. For GEMM the DUE rate is extremely low for all devices as the code has little control-flow operation, which reduces the chances of having a data corruption to lead to a Crash. Moreover, the scheduling for GEMM is very simple and defined, reducing the strain on the scheduler and, thus, the probability of corruption.

The SDC/DUE ratio is higher for the ARM, Maxwell, and Steamroller (between 1.5 and 5 SDCs for each DUE) than for the Kepler with ECC OFF and Pascal (between 2 and 11 SDCs for each DUE). This SDC/DUE ratio is explained considering that the host is exposed for embedded devices, while for GPUs and KNC the host is out of the beam. An error during host operations is likely to create a DUE. The comparison between Figure 7.2 and Figure 7.4 shows that for most algorithms (but not for sort) DUEs are more likely to be generated by radiation-induced faults in control-logic or inaccessible resources rather than by data corruption. DUEs are more likely for sort algorithms than for other codes only for unprotected devices (ARM and Kepler with ECC disabled). As explained in the following, for sort, ECC can prevent data corruption-induced DUEs. Moreover, KNC and Kepler with ECC enabled have, on the average, 1.2 and 4.4 DUEs for every SDC, respectively. This is because ECC reduces SDCs but could increase DUEs.

An additional interesting insight that can be retrieved from data in Figure 7.3 is that ECC reduces significantly the SDC rate but not the DUE rate. Among the tested devices only the Kepler and the KNC implement ECC. The ECC reduces the Kepler SDC FIT of about one order of magnitude. The reduction is accentuated for codes that heavily use memory, like Quick and Merge sort. Unfortunately, KNC ECC cannot be disabled, preventing the measure of its efficacy. However, an empiric sign that ECC is effective for the KNC is that its SDC FIT rates are comparable (if not lower) than much smaller devices for all of the tested codes but Hotspot. For Quick sort, the KNC SDC FIT rate

Figure 7.5: Mean Workload Between Failure: amount of useful data produced before experiencing a SDC or DUE. Higher values imply higher reliability.



Source: The Author

is even (significantly) smaller than the ARM one. As already mentioned, Hotspot has a different trend as it uses much less memory than other codes, reducing ECC benefits. For the Kepler, enabling ECC reduces Hotspot SDC rate of only 70%, while for the other codes the reduction is at least of one order of magnitude.

As shown in Figure 7.4, ECC increases the DUE rate for most codes. This increase is because (1) ECC triggers an application crash when it detects an uncorrectable error and (2) logic resources are left unprotected, preventing the mitigation of faults affecting scheduler and dispatcher. ECC reduces the DUE only for sorting algorithms. As sorting uses a lot of data-based control flow decisions (indexes, pivot elements), protecting memory is likely to reduce the DUE rate significantly. The ECC efficacy for sorting is in accordance with fault-injection data for Kepler and KNC in Figure 7.3 that shows a high percentage of injections in registers to produce DUE. ECC masks those faults, reducing DUE rate.

Comparing the SDC rate of GEMM, LavaMD, and Hotspot as executed on the Kepler with ECC disabled (CMOS) with Pascal and KNC (FinFET) demonstrates that FinFET transistors play a significant role in reducing the probability of errors. As Kepler and Pascal's architectures are similar and the executed CUDA source code is exactly the same for the two devices, they are likely to have a similar AVF and PVF. On the contrary, as illustrated in Section 7.3, KNC has a significantly different way of propagating faults. Thus, we limit the CMOS-FinFET comparison on Kepler with ECC disabled (Pascal does not have ECC) and Pascal. As shown in Figures 7.3 and 7.4, even if Pascal has a higher amount of resources (details in Section 7.1.1), both Pascal's SDC and DUE rates are, on the average, one order of magnitude lower than Kepler's with ECC OFF, for all the tested codes.

These results represent the first proof on live hardware of claims in recent publications, showing through test chips lower error rates for FinFET transistors compared to CMOS (NOH et al., 2015). Additionally, FinFETs are much more effective in reducing DUEs (Figure 7.4) and the SDCs of compute-intensive codes (i.e., LavaMD and Hotspot) versus the Kepler CMOS device, supporting the advertised benefits of FinFET technology in terms of reducing errors in logic versus errors in memory (WANG et al., 2006).

Transistor implementation also explains why, despite the higher amount of resources, Steamroller SDC and DUE FIT rates are, on the average, 50% and 40% the Maxwell ones, respectively. Steamroller, in fact, is built with $28nm$ CMOS and Maxwell with $20nm$ CMOS transistors. Planar devices are known to be less reliable as transistor dimensions shrink (BAUMANN, 2005).

7.2.3 Mean Workload Between Failures

A higher number of resources increases the FIT rates. However, the additional resources used for computation could bring higher throughput, increasing the amount of useful data produced before experiencing a failure. To consider also resources utilization efficiency we measure the MWBF, which indicates the amount of useful data correctly processed before experiencing a failure. MWBF is inversely proportional to FIT and directly proportional to the throughput (measured dividing the data produced by execution time listed in Table 7.2).

Figure 7.5 shows the MWBF measured for all the devices and codes. Interestingly, comparing Figures 7.3 and 7.4 with Figure 7.5, we notice that higher FIT rate does not imply lower MWBF (and, thus, lower reliability). The fact that MWBF follows a different trend than FIT rates means that the throughput of the tested codes is more dependent on the device than the FIT rates. The ARM processor is an extreme case, as it shows one order of magnitude lower FIT rate compared to the Kepler. However, Kepler shows an MWBF three orders of magnitude higher compared to the ARM and is, then, much more reliable in terms of produced correct data. The benefit of the efficient use of the additional resources available on the Kepler is higher than the increased error rate they bring. This benefit is particularly true for computing intense algorithms, like GEMM and Hotspot, executed on GPUs for which the additional memory available is used to speed up executions, reducing memory latencies (and, thus, FIT) while increasing the throughput. Comparing Figure 7.5 and Figures 7.3 and 7.4 we can conclude that, for

the tested configurations, devices with higher FIT rates are the ones with higher MWBF (i.e., they are the more reliable). We believe this to be a promising and generic result: when additional resources are efficiently used, the MWBF is likely to increase. When the device utilization is increased, the exposed area (and thus FIT rate) increases linearly. As architectures and codes are typically designed to use the available resources efficiently, a higher resources availability eventually improves super-linearly the throughput.

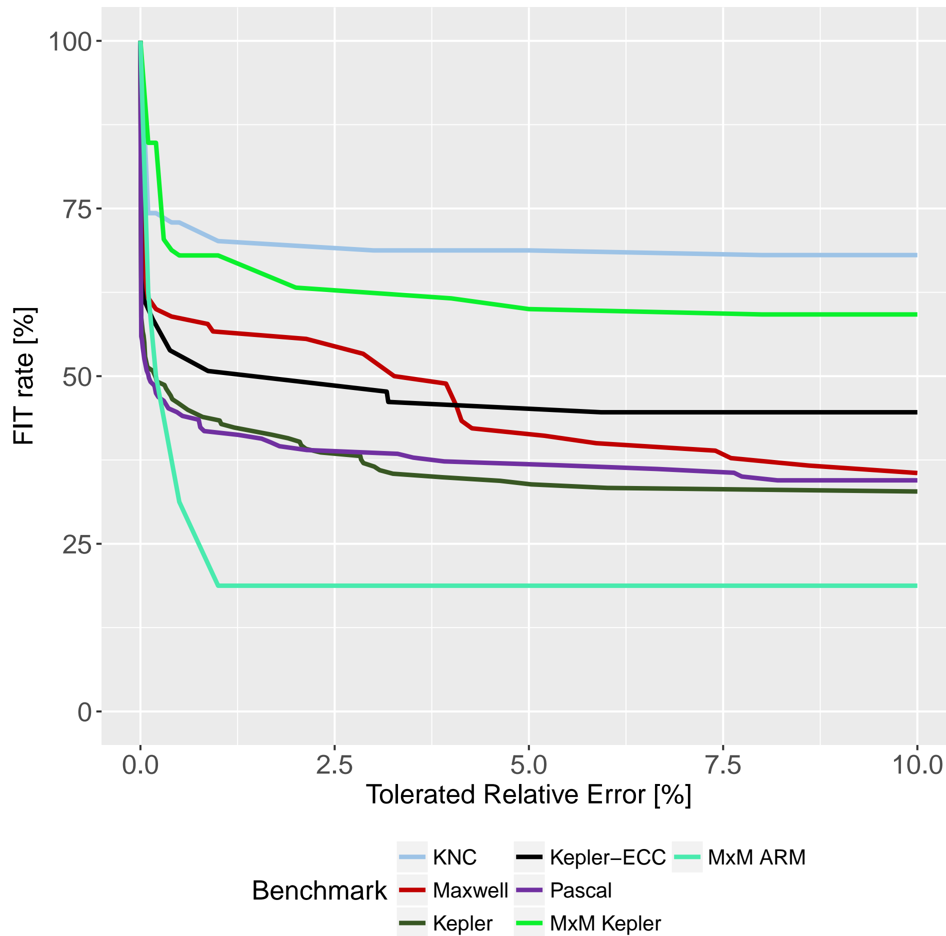
MWBF also underlines the benefits brought by ECC. As the throughput of the selected codes is not significantly affected by ECC (but could affect codes that heavily uses DDR, as the ECC reduces its bandwidth), the MWBF benefits from the reduced SDC rate ECC provides, increasing the Kepler reliability significantly. A surprising result is given by Hotspot, for which Pascal (which, we recall, does not have ECC) has the highest MWBF. As said, Hotspot uses less memory per operation than other codes reducing ECC benefits. On the contrary, FinFET transistors are even more robust to radiation than CMOS for logic elements versus memory (WANG et al., 2006), making Pascal the more reliable device for extremely parallel and compute-bound codes.

7.3 Error Criticality Analysis

In this section, we aim at qualifying the observed SDCs regarding the difference between the corrupted output and the expected, error-free, output. We consider how different the values of the corrupted elements are from the expected ones and how many elements of the output are corrupted. The way errors manifest at the application output is a symptom of how errors propagate inside an architecture and through the code. The proposed qualifications are then used to detect code-dependent and device-dependent error propagation behaviors.

Figures 7.6, 7.7, 7.8, and 7.9 show the FIT rate reduction for GEMM and MxM, Hotspot, LavaMD, and FFT as a function of the accepted output value approximation. We show how varying the acceptable approximation margin (i.e., how different from the expected values the elements can be to accept the execution as correct) affect the SDC FIT rate for each benchmark. In other words, increasing the acceptable error margin (horizontal axis) the FIT rate can decrease (vertical axis) as some errors fall into the approximation margin. It is worth noting that we remove a faulty execution from the FIT rate count only when all its corrupted elements are inside the approximation margin. Sorting algorithms are treated separately. As NW is executed with integer values, the

Figure 7.6: GEMM relative error reduction.



Source: The Author

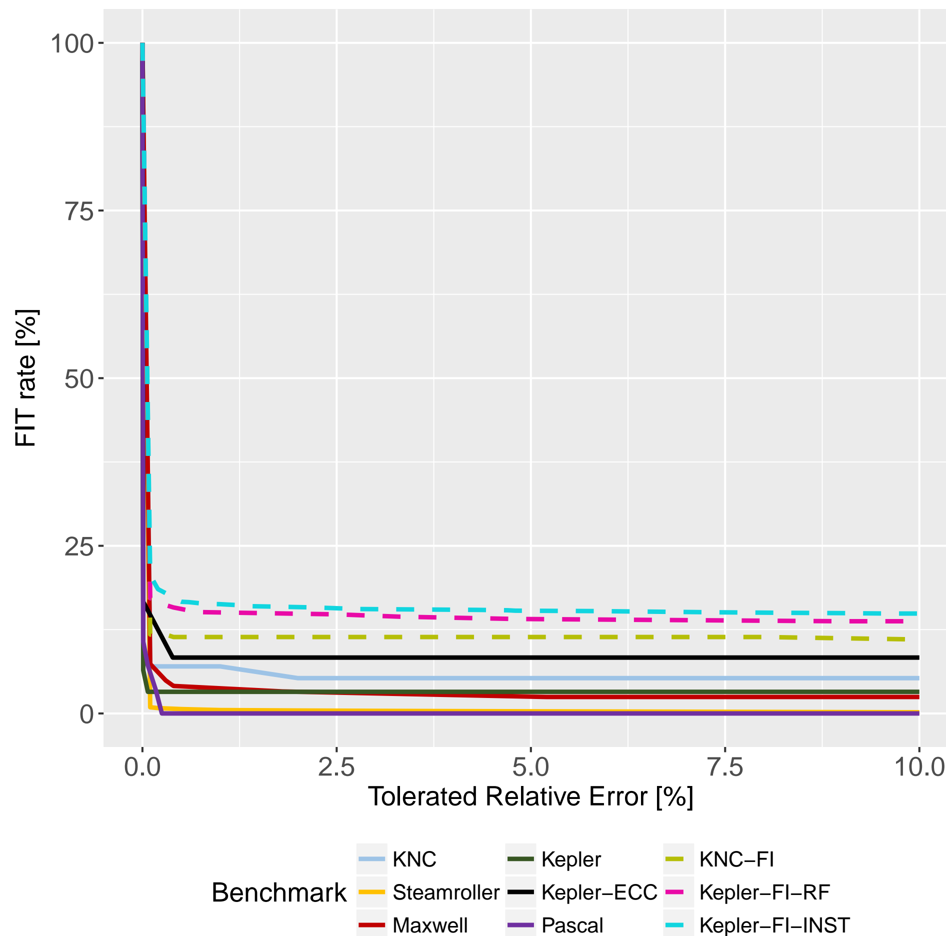
approximation is less interesting and not shown.

Figures 7.10, 7.11, and 7.12 show the percentage of faulty executions that are affected by a single or multiple corrupted elements. When multiple errors occur, we plot the portion of the output elements that are corrupted. We classify the number of corrupted elements between 1% and 9% of the output elements. 10% or more are grouped. It is worth noting that multiple corrupted elements are generated by the interaction of a single neutron.

Device-Dependent Behaviors:

Figures 7.6, 7.7, 7.8, and 7.9 show that, independently on the code, KNC has the lowest FIT rate reduction while ARM has the highest reduction. GPU architecture has a FIT rate reduction between KNC and ARM. KNC reduction is lower because an error generates more corrupted elements in the output, as seen in Figures 7.10, 7.11, and 7.12. Then, a higher number of corrupted elements may lead to elements with a higher relative error. KNC tends to spread the error to more elements in the output because of

Figure 7.7: *HotSpot* relative error reduction.



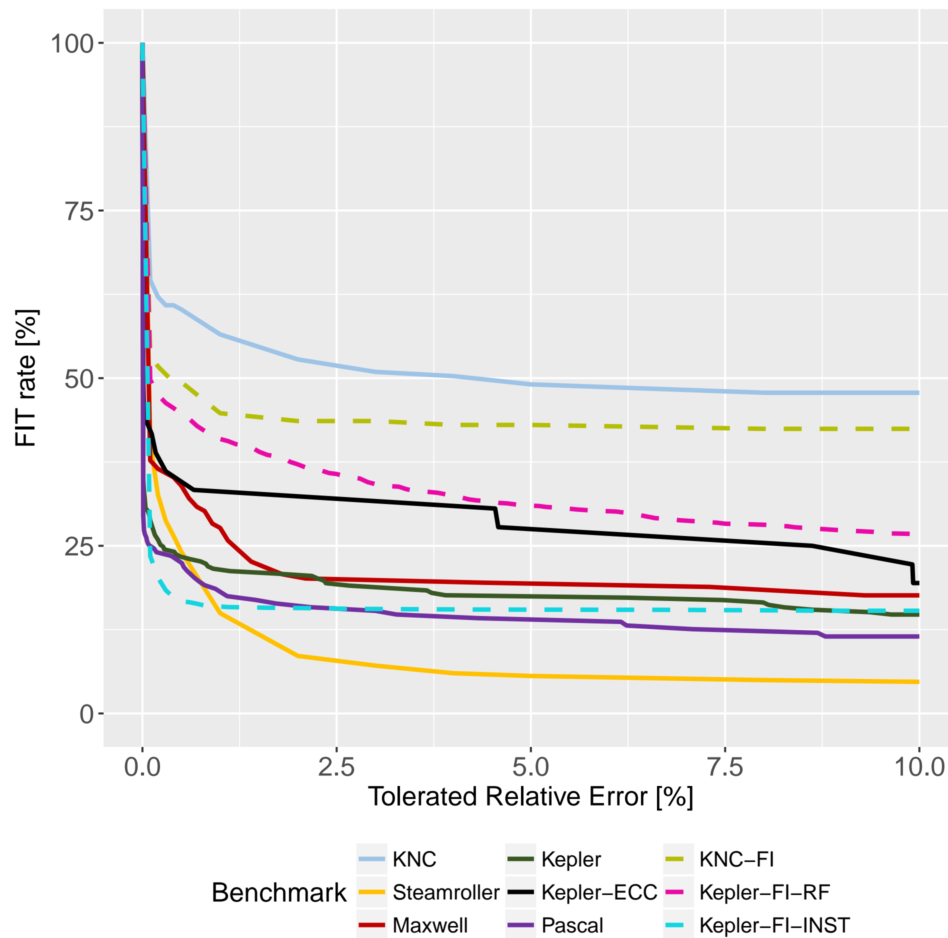
Source: The Author

the memory management system. The memory system treats an error localized into one of the KNC cores as a valid update, which will propagate to the other cores and main memory by the cache coherency protocol and automatic write-back. Since GPUs have a less transparent memory management system, localized errors will not be automatically propagated, restricting the number of corrupted elements. ARM executes the benchmarks in bare-metal and sequentially, thus, ARM shows a different error propagation than KNC and GPUs presenting a high number of single element corruption.

Kepler with ECC ON has a smaller FIT reduction compared to ECC OFF. While the absolute number of SDC is significantly reduced (as discussed in Section 7.2.2), ECC corrects mainly errors that have a smaller difference from the expected value. Our analysis suggests that fault in unprotected resources are, unfortunately, the ones that most affect the output. ECC is likely to also impacts the lower reduction for KNC.

Code-Dependent Behaviors: Figures 7.6, 7.7, 7.8, and 7.9 show that there is also a significant code-dependent component in the FIT rate reduction. When a 5% of output

Figure 7.8: *LavaMD* relative error reduction.

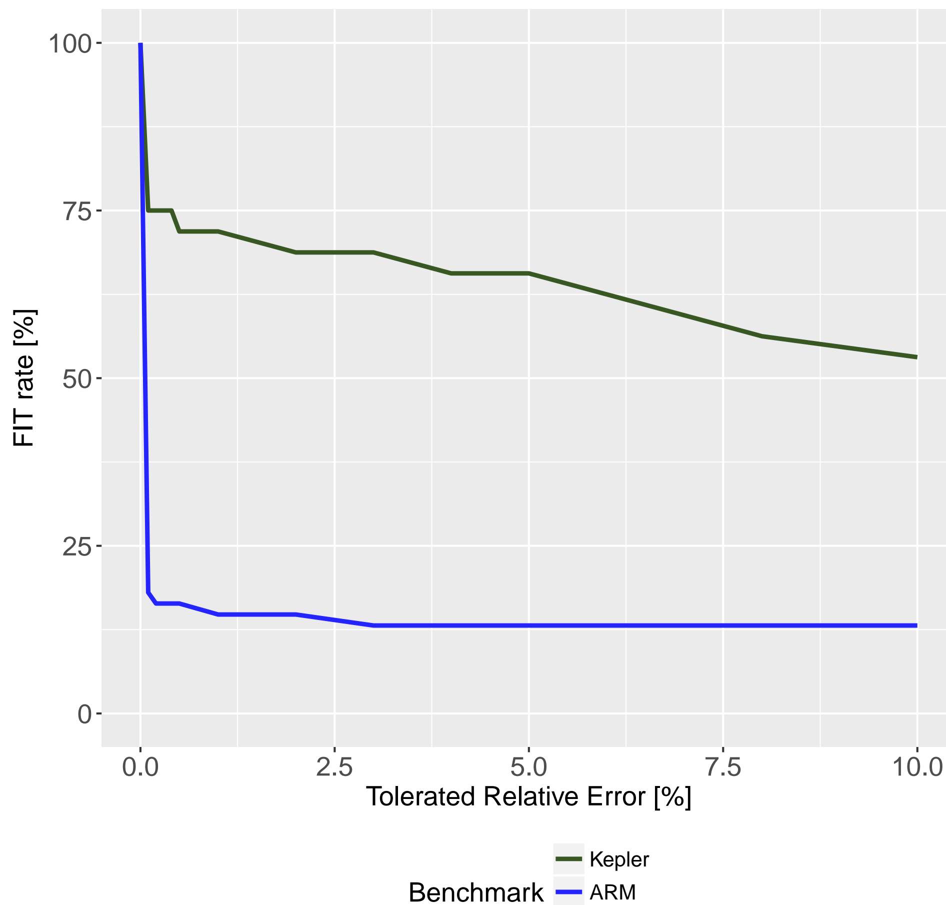


Source: The Author

approximation is tolerated, GEMM, MxM, and FFT show the smaller FIT rate reduction, 30% to 75%. LavaMD has a reduction of 50% to 90%. Finally, Hotspot benefits from a FIT rate reduction of more than 90%. We analyze each code separately.

Figure 7.6 shows the FIT rate reduction for **GEMM and MxM**. Both benchmarks present the lowest FIT rate reduction despite presenting the highest percentage of single errors (see Figures 7.10, 7.11, and 7.12). GEMM and MxM are purely arithmetic codes, while LavaMD and Hotspot are iterative simulations. For LavaMD and Hotspot, as explained in the following, a corrupted element during computation could be smoothed (and spread) interacting with neighbors. As a result, a fault could, in the worst case, interact with all the values that contribute to the calculation of all the output elements. On the contrary, for GEMM and MxM once a value has been corrupted during computation, it will contribute to the evaluation of, at most, $2N$ output elements. The chances of spreading are then lower than for other codes. Additionally, if matrices are dense (as the ones used for our test), the error is likely to accumulate during computation, reducing the possibility

Figure 7.9: *FFT* relative error reduction.



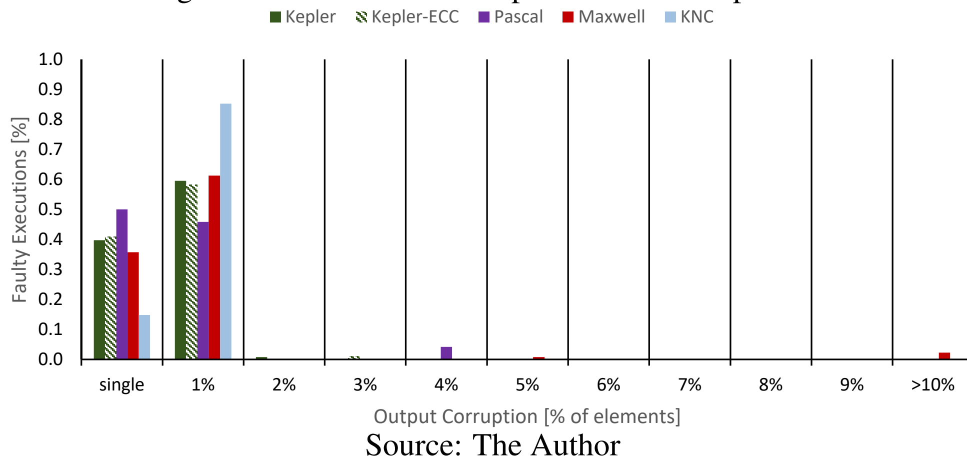
Source: The Author

of having the error smoothed.

Hotspot has the more representative FIT reduction, shown in Fig 7.7. Hotspot is an iterative stencil algorithm. As the program output tends to reach an equilibrium, the elements interact with each other to simulate the temperature profile of a surface. An error that increases the temperature of a pixel (or of some pixels) significantly will cause the temperature to be averaged with neighbors pixels. As equilibrium is reached, errors will then dissipate and be smoothed throughout further neighbors. Figure 7.11, in fact, shows that single corrupted elements are extremely rare for Hotspot and Fig 7.7 shows that Hotspot leads to very low relative error. Even tolerating only 0.2% of relative error can reduce at least 90% of the FIT, regardless of the architectures in which the code is executed. This is a promising result for approximate computing. We can expect stencil applications to be extremely resilient even if a small margin of output value approximation is accepted.

LavaMD FIT rate reduction is depicted in Figure 7.8. KNC presents the lower

Figure 7.10: *DGEMM* corrupted elements dispersion.



Source: The Author

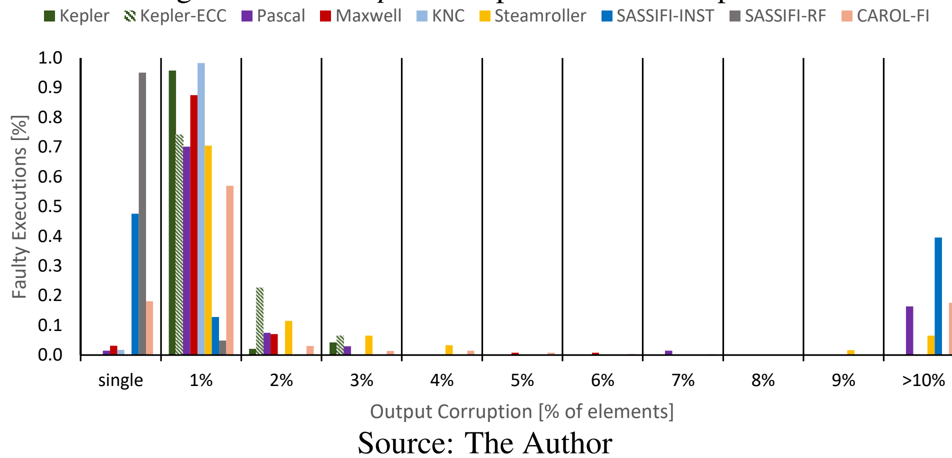
FIT rate reduction, saturating at 50% as we increase the acceptable error margin till 10%. Steamroller shows the highest reduction, saturating at 5%. The FIT rate reduction for LavaMD follows the same trend as the number of corrupted elements in the output (refer to Figure 7.12). For KNC, we never observed any single corrupted elements, while Steamroller has about 50% of faulty executions affected by a single corrupted element. NVIDIA GPUs presents a lower number of single corrupted elements than Steamroller, leading to a slightly lower FIT rate reduction. In contrast to GEMM, LavaMD error effect depends on the error magnitude, simulation time, particle position, and neighboring particles. Therefore, LavaMD error effect can be negligible or generate a cascade of interactions that will move the result further away from the expected one.

FFT FIT rate reduction can be seen in Figure 7.9. Kepler reaches about 50% of FIT rate reduction. Similar to GEMM and MxM, an error will likely accumulate during the execution than being smoothed out, explaining a similar reduction. We do not show the output corruption for FFT as all the corrupted output were affected by less than 1% of corrupted elements. As expected from the analysis made so far, ARM shows single corrupted elements while the Kepler shows mostly multiple corrupted elements.

CAROL-FI and SASSIFI fault-injection results are also included for Hotspot and LavaMD (Figures 7.7, 7.11, 7.8, and 7.12). Fault injection correctly estimates the magnitude of corrupted elements in the algorithm output for each architecture. Thus, fault injectors proved to be a valuable tool to study approximate computing and its effect on radiation sensitivity.

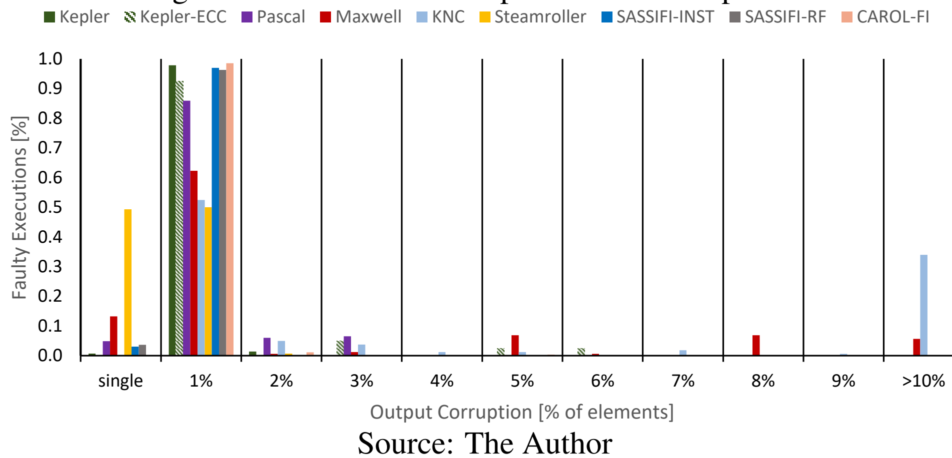
The difference between the corrupted and expected values are not useful for **Quick sort** and **Merge sort**. We only show, in Figures 7.13 and 7.14, the portion of the output array that is affected for Quick sort and Merge sort, respectively. We count as errors the

Figure 7.11: *HotSpot* corrupted elements dispersion.



Source: The Author

Figure 7.12: *LavaMD* corrupted elements dispersion.

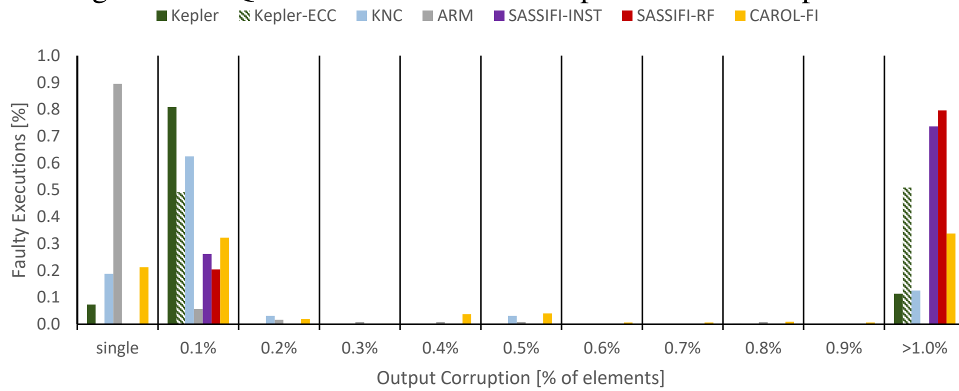


Source: The Author

number of corrupted or missing elements as well as the number of unordered elements. Executions with multiple output elements corruption are classified by the percentage of output corruption up to 1%, while >1% groups the ones that exceed 1%.

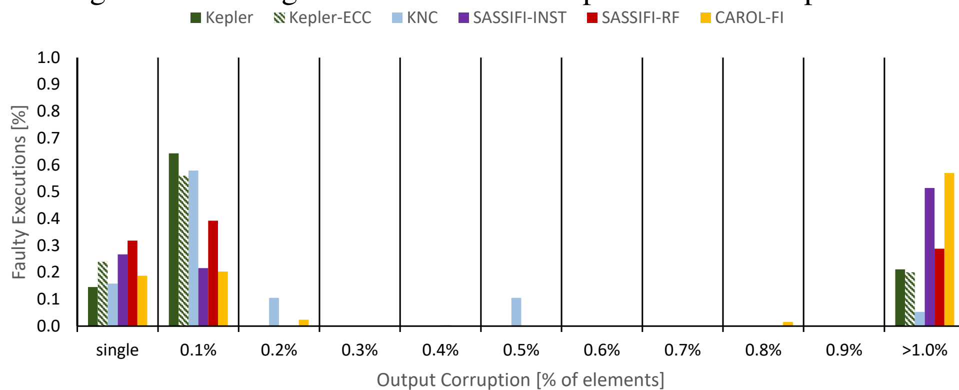
As depicted in Figures 7.13 and 7.14, consistently to other codes, ARM presents the lowest percentage of output corruption, since more than 85% of SDCs produce a single output element corruption. Unlike parallel architectures, ARM has a very limited resource sharing during computation. In the ARM, each couple of elements is calculated separately from others, thus reducing failure propagation through the algorithm when compared to parallel architectures. For Quick sort, ECC reduces mostly the errors that have a lower impact on application correctness, confirming the trend from other codes. When ECC is enabled on the Kepler, there are no single errors and more than 50% of executions have more than 1% of the elements affected. On the contrary, the trend with ECC ON and OFF is very similar to Merge sort. Memory reuse, typical of divide-and-conquer algorithms, makes errors to spread easily, even without ECC.

Figure 7.13: Quick sort number of corrupted elements dispersion.



Source: The Author

Figure 7.14: Merge sort number of corrupted elements dispersion.



Source: The Author

It is interesting to notice that the trend for KNC and Kepler are very similar for both Quick and Merge sort: about 70% of corrupted executions show few elements corrupted. Fault-injection, on the contrary, produces a corruption that affects a higher portion of the output array.

7.4 Discussion

We have compared the reliability of an extensive set of devices and codes measured through both beam radiation experiments and, whenever available, fault-injection. The results data highlights both code-dependent and device-dependent behaviors. We found that the DUE rate is mostly device-dependent and cannot be easily predicted using fault-injection. This work also shows that an efficient use of a higher amount of resources justifies the increase in FIT rate. Moreover, FinFET transistors are effective in reducing both the SDC and DUE rates independently of the code, while ECC is more effective in memory-bound codes. Softwares with long memory latencies have a higher SDC rate, es-

pecially when ECC is not available. Moreover, additional resources, although increasing the FIT rates, generally improve MWBF.

The main generic insights we derive from the reliability analysis can be summarized as follows:

Device-dependent behaviors:

- The DUE rate is device-dependent and cannot be easily predicted using fault-injection.
- FinFET transistors are effective in reducing both the SDC and DUE rates, independently on the code.
- Regardless the executed code SDCs are more frequent than DUEs if there is no ECC.
- The tested configurations with higher FIT have higher MWBF than the others. Additional resources increase throughput more than FIT rate.

Code-dependent behaviors:

- SDC rate is mainly code-dependent. Fault injection helps in identifying codes with higher SDC rate.
- For memory-bound codes, ECC reduces the SDC rate significantly and slightly the DUE rate. For compute-bound codes, ECC still reduces SDC but harms DUE.
- Codes with long memory latencies have a higher SDC rate, especially when ECC is not available.
- Compute-bound codes exacerbate the benefits, in terms of MWBF, that additional resources bring.

We have also qualified the observed errors in terms of the difference between the expected value and the observed one, and the percentage of the output that has been corrupted. Stencil applications, independently of the device, could take great advantage of approximate computing as 90% of corrupted executions could be accepted as correct with a 0.2% of output approximation. While KNCs spread errors because of memory coherency, GPUs corrupt fewer elements. Finally, ECC is shown to correct mainly errors that have a lower impact on the output.

8 HARDENING SOLUTIONS

Software-based solutions may be the only way to protect the system when using COTS devices. COTS hardware cannot be changed, and the hardware-based hardening solutions provided by the device may not be enough to achieve the required level of resilience. Then, in this chapter, software-based hardening solutions will be implemented and evaluated alongside hardware-based hardening solutions available. Finally, this chapter also presents a selective hardening approach and shows the improvements such technique can yield.

8.1 Hardware-based vs Software-based Hardening

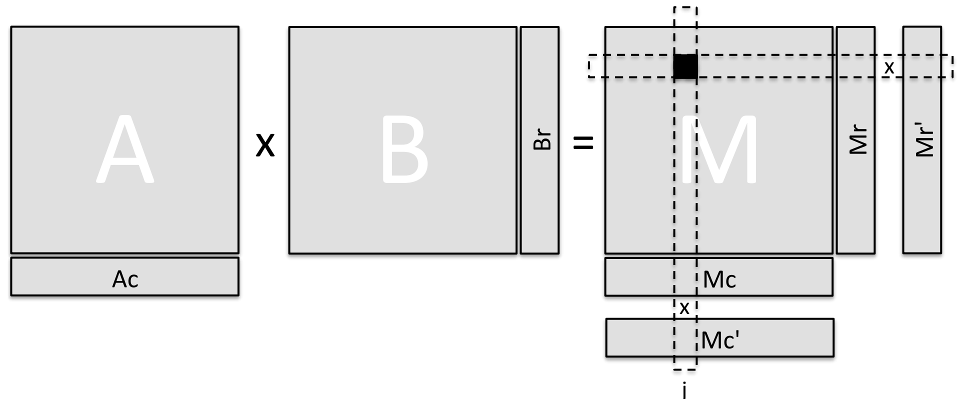
NVIDIA GPUs provide a hardware implementation of ECC mitigation mechanism that can be disabled. Therefore, we use this feature to evaluate the ECC against two software-based hardening techniques without the influence of the hardware mechanism. We use a specific and a generic technique to assess how well software techniques can perform regarding execution time and resilience. Intel Xeon Phi also provides hardware mitigation mechanisms, but the mechanisms cannot be disabled making it hard to measure the impact of software techniques without the influence of the hardware ones.

For this section, we use the NVIDIA K20 device described in section 4.1.2 since it is the same device as the authors of the specific techniques used in (RECH et al., 2013a; PILLA et al., 2014). Thus, we can compare the ECC and the generic technique we implemented in the same device.

8.1.1 Error-Correcting Code

The ECC mechanisms included in advanced GPUs can correct single bit errors and detect double-bit errors on the main memory structures (NVIDIA, 2012). The ECC included is the only mitigation mechanism available for HPC GPUs. When enabled, the ECC reduces the DDR availability of about 15% (NVIDIA, 2012). The results, presented in section 8.1.4, show that enabling ECC reduces the SDC FIT by about one order of magnitude for all the tested algorithms. Some SDCs still occur even if ECC is enabled as flip-flops, queues, logic resources, and schedulers are left undetected. The former

Figure 8.1: Algorithm-Based Fault Tolerant for matrix multiplication (HUANG; ABRAHAM, 1984).



Source: (RECH et al., 2013a)

resources details are considered business-sensible, and are not available, while the latter resources have been demonstrated to contribute significantly to the GPU SDC rate (RECH et al., 2014).

The main issue with ECC is that even if the instruction cache and registers are protected (NVIDIA, 2012), the GPU becomes more prone to experience a DUE, imposing a higher checkpoint frequency in HPC applications. When ECC is off, an incorrigible MBU may be masked during computation, without appearing at the output (WILKENING et al., 2014). Nevertheless, that same MBU would be detected by the ECC as an incorrigible error, triggering the application DUE. Additionally, the ECC will not reduce the DUE caused by scheduler corruptions as those resources are left unprotected.

8.1.2 Algorithm-Based Fault Tolerance

Since ABFT strategies are algorithm-specific, to exemplify its use two different applications were hardened: a Matrix Multiplication and a Fast Fourier Transform.

8.1.2.1 Matrix Multiplication

The matrix multiplication application was hardened using the ABFT strategy proposed by Huang and Abraham (HUANG; ABRAHAM, 1984), which is based on the result checking approach of Freivalds (FREIVALDS, 1979). Input matrices A and B are coded before computation, adding column and row checksum vectors (A_c and B_r in Figure 8.1) by summing all the elements in the correspondent column or row.

The result of the multiplication of the expanded matrices is a fully-checksum ma-

trix M, where the $(n+1)$-th row and the $(n+1)$-th column contain the column (M_c) and row (M_r) checksum vectors of M, respectively (FREIVALDS, 1979). When the multiplication is finished, M column and row checksum vectors are re-calculated summing the first n columns and n rows of M, resulting on M'_c and M'_r, respectively. Output verification is done by comparing the checksum vectors from the multiplication and the newly computed ones.

If a mismatch is detected between $M_r[i]$ and $M'_r[i]$, it means that at least one error is present in the i-th row of M, and respectively for columns. If $M[i,j]$ is identified as the only error in M, it can be corrected quickly using either the row or column checksum vectors following Equation 8.1 or Equation 8.2 (HUANG; ABRAHAM, 1984).

$$M_{correct}[i,j] = M[i,j] - (M'_r[i] - M_r[i]) \tag{8.1}$$

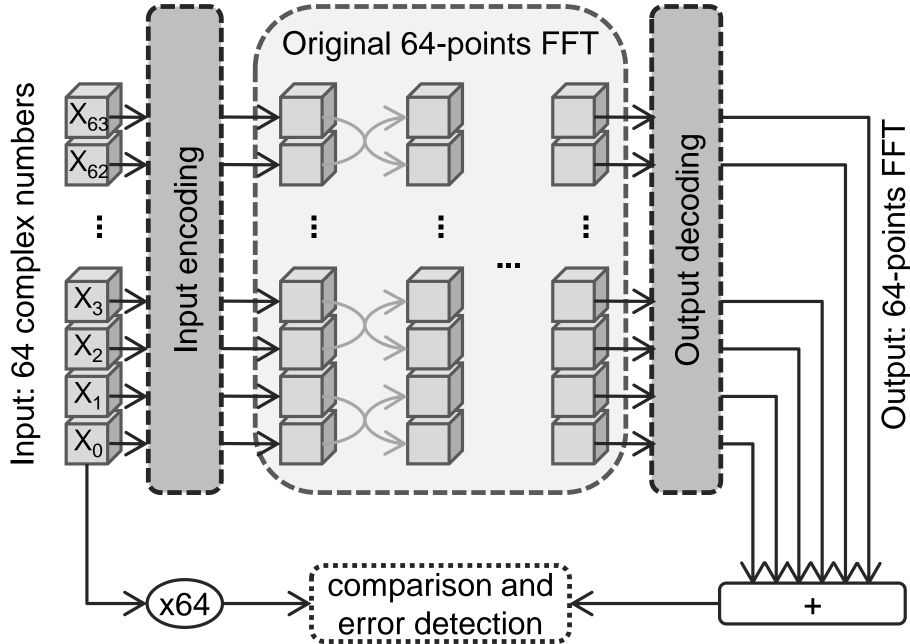
$$M_{correct}[i,j] = M[i,j] - (M'_c[j] - M_c[j]) \tag{8.2}$$

On a GPU, the operations required to compute the checksums and detect errors can be done in $O(n)$, while error correction takes constant time (RECH et al., 2013a). The technique proposed by Huang and Abraham is only capable of correcting single output errors (HUANG; ABRAHAM, 1984), which have been experimentally demonstrated to correspond to less than 43% of the cases (RECH et al., 2013a). Thanks to radiation experimental data, the ABFT strategy was extended to correct multiple errors on the same row or column of M in constant time and randomly distributed errors in $O(e_r \times e_c)$, where e_r and e_c are the number of mismatches between M row and column checksums, respectively (details in (RECH et al., 2013a)). Please note that the ABFT implementation and the reliability evaluation of Section 8.1.4 are neither dependent on implementation (MxM or ***DGEMM***) nor on input size.

8.1.2.2 Fast Fourier Transform

Fast Fourier Transform application was hardened using the ABFT strategy proposed by Jou and Abraham (JOU; ABRAHAM, 1988) for fault-free N-point FFT networks of N processors (PILLA et al., 2014). This strategy is based on the superposition principle of linear systems and the circular shift property of the FFT. Its basic idea is to detect errors arising in any processor or connection with the use of input coding and a checksum comparison at the output. Figure 8.2 illustrates this process. The ABFT

Figure 8.2: Algorithm-Based Fault Tolerant FFT. The 64 complex elements of the input are coded, then the 64-point FFT is performed with the original algorithm, and its output is decoded. Errors are detected comparing the checksum generated summing the output values with $X_0 \times N$.



Source: (PILLA et al., 2014)

was implemented on 512×512 64-point FFTs. Nevertheless, the reliability evaluation presented in Section 8.1.4 is easily extendable to other input sizes.

A thread in the hardened algorithm starts by taking its input sequence of N complex elements X and encoding it, resulting in the sequence X', as represented in Equation 8.3. The original vector X is kept unmodified for the situation where the FFT has to be re-computed due to the detection of an internal error. After the encoding phase, vector X' is given as input to the original FFT algorithm. After the algorithm computation is completed, the FFT output stored in X' is decoded to vector Y following Equation 8.4, where w_N^{-k} is the Nth root of unity. The N decoded results are then summed, generating a checksum. As formally demonstrated by Jou and Abraham (JOU; ABRAHAM, 1988), this encoding and decoding scheme gives each output a nontrivial weighted contribution to the checksum such that any error will cause a detectable error syndrome, which is not the case with the original FFT algorithm. After computation, the checksum is compared to $N \times X[0]$. Any mismatches will identify the FFT as faulty and will signalize the necessity of a re-computation. It is important to notice that both encoding and decoding phases

could be processed by more than one thread for each 1D FFT, increasing parallelism.

$$X'[i] = \begin{cases} 2X[i] + X[i+1] & 0 \leq i < N-1 \\ 2X[i] + X[0] & i = N-1 \end{cases} \tag{8.3}$$

$$Y[k] = \left\{ \frac{1}{2+w_N^{-k}} X'[k] \quad 0 \leq k < N \right. , \tag{8.4}$$

In addition to encoding and decoding data, our ABFT strategy also includes two vectors to signalize any FFT re-computations needed, and to detect any failures from SM interruptions and scheduler failures that could prevent threads from completing their execution. Overall, this hardening mechanism increases the GPU memory usage with the addition of the encoded matrix X' and these two vectors (which are much smaller than the former). Lastly, the proposed ABFT strategy keeps the computational complexity of the original algorithm, as the functions for input encoding, output decoding, and error detection are linear. In addition, only the y n-point FFTs with errors are re-computed, and their overhead ends hidden by the parallel execution of the other $512 \times 512 - y$ FFTs (PILLA et al., 2014).

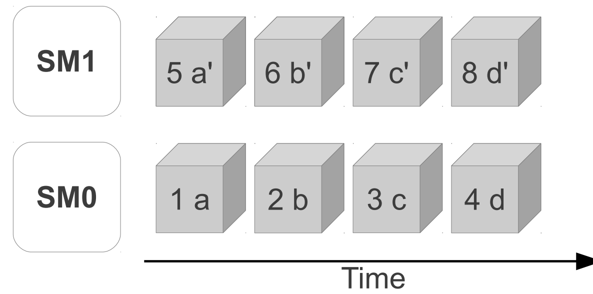
8.1.3 Duplication With Comparison

DWC can be implemented in several ways on a GPU, such as by duplicating blocks, threads, or by executing operations twice in a thread. It is worth noting that duplicated threads must be carefully distributed as the corruption of shared resources (like caches) or critical resources (like the scheduler) may propagate to both copies, undermining DWC detection capabilities.

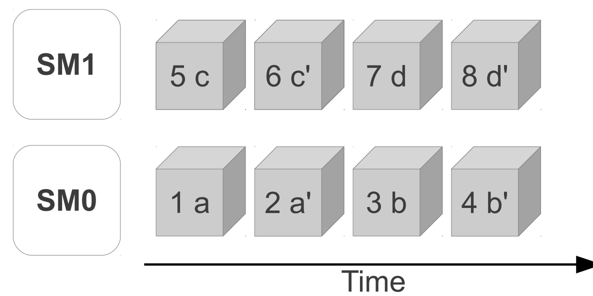
We designed three Duplication With Comparison strategies with different duplication and distribution philosophies: *Spatial*, *E-O Spatial*, and *Time*. In the spatial duplication strategies (*Spatial* and *E-O Spatial*), the number of blocks needed to compute the solution and, thus, the number of instantiated threads is doubled. The thread blocks are divided into two domains, each executing the unhardened code. The result of a block is then compared to the result of the duplicated block, detecting possible mismatches. We choose to duplicate blocks instead of threads in a block to avoid having both instances using the same cache, as a radiation-induced failure in it would be likely to propagate to both copies undetected.

In *Spatial*, a domain is formed by the first half of the blocks and the other domain

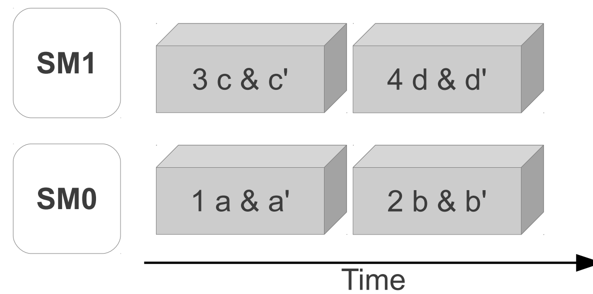
Figure 8.3: Duplication With Comparison strategies. Each block is represented by a box and have its identification index (i.e., *block ID* in CUDA) and a letter corresponding to the task assigned to it. The apostrophe after a letter means that the task is a duplicated one.



(a) Spatial



(b) Even-Odd Spatial



(c) Time

Source: The Author

by the second half. The redundant blocks are not necessarily executed in parallel with the first half of the blocks when the number of blocks instantiated in the unhardened code already exceeds the GPU parallel capabilities. The second spatial duplication, *E-O Spatial*, alternates original blocks and duplicated blocks in an even-odd fashion. In other words, blocks i and $i+1$ execute on the same data. *Spatial* and *E-O Spatial* are illustrated in Figure 8.3a and Figure 8.3b.

Spatial and *E-O Spatial* detection capabilities may significantly differ. The redun-

dant calculations are more likely to be processed in the same SMs in *E-O Spatial*, sharing resources like caches. Even if more efficient, as data does not have to be moved in two different regions of the GPU, *E-O Spatial* may experience errors in the SM caches that propagate to both the duplicated blocks and thus remain undetected. On the other hand, the same data is going to be duplicated in two different locations of the GPU in *Spatial*, reducing performance and increasing the exposed area of the device.

In the last DWC strategy implemented, named *Time* and depicted in Figure 8.3c, each thread performs a redundant calculation after completing the original one. In *Time*, about the same amount of parallel resources (i.e., number of threads, scheduling strain, caches) as the original implementation are employed. However, threads complexity is increased as each thread executes twice the operations and compares results.

When an error is detected, the faulty threads are re-executed in the GPU until the results of the execution and the redundant execution match, or a maximum number of re-executions is reached resulting in a failed execution. In our experiments, each block or thread will be re-executed a maximum of 5 times when an error is detected. Nevertheless, all errors detected by our strategies were corrected in the first re-execution.

8.1.4 Evaluation

In this section, the efficiency and efficacy of software-based techniques are evaluated and compared to the ECC available in modern high-end GPUs. The results shown here were obtained using the GPU device described in section 4.1.2. Results for ABFT were obtained using 2048×2048 matrices for MxM and 512×512 64-point FFTs, while DWC strategies were applied to *HotSpot*. Nevertheless, similar error correction/detection capabilities are expected for different input sizes and algorithms.

Table 8.1 reports the reliability obtained for the unhardened code, the ECC, and the software-based techniques. For SDC and DUE, we present two columns for each one; the first one is the absolute FIT and the second one present the FIT in arbitrary unit (a.u.), which normalizes the FIT of hardened code to the unhardened version to facilitate comparison. The last column of Table 8.1 reports the execution time of the hardened codes normalized to their unhardened versions.

When ECC is enabled, a 55% increase in DUE FIT is observed for MxM, 44% for FFT, and 43% for $Hotspot$. SDC and DUE results for ECC were similar for the tested algorithms, and are expected to be comparable also for other codes. Enabling ECC affects

Table 8.1: Efficiency and resiliency of the available hardening solutions for GPUs.

		SDC		DUE		Perf.
		FIT	a. u.	FIT	a. u.	
MxM	Unhardened	5.79×10^2	1.000	2.64×10^2	1.000	1.00
	ECC	5.62×10^1	0.097	4.02×10^2	1.523	1.01
	ABFT	1.04×10^1	0.018	2.97×10^2	1.127	1.14
FFT	Unhardened	2.88×10^3	1.000	7.02×10^2	1.000	1.00
	ECC	4.14×10^2	0.144	1.01×10^3	1.435	1.50
	ABFT	5.18×10^1	0.018	8.04×10^2	1.145	1.18
$Hotspot$	Unhardened	2.04×10^2	1.000	1.12×10^2	1.000	1.00
	ECC	1.81×10^1	0.089	1.61×10^2	1.439	1.00
	Spatial	0.00×10^0	0.000	1.05×10^2	0.937	2.51
	E-O Spatial	3.26×10^0	0.016	8.40×10^1	0.750	2.45
	Time	2.45×10^0	0.012	8.85×10^0	0.079	1.90

the execution time of FFT significantly, while it barely affects MxM and $Hotspot$. This execution time loss is explained noting that FFT requires continuous accesses to the GPU DDR. When ECC is enabled, the data transfer bandwidth from and to the DDR is reduced as code data has to be transferred as well. If the unhardened version of the code saturates the bandwidth, as in the FFT case, the execution time is then likely to be higher when ECC is enabled.

The ABFT FFT shows an SDC FIT that is almost one order of magnitude lower than the ECC protected version, while the ABFT MxM has an SDC FIT which is less than 18% the ECC protected one. The ABFT mechanisms are then more effective than ECC in increasing the GPUs resilience, since they are designed to detect mismatches in the output independently of their causes while, as said, ECC only corrects errors on memory elements directly affected by radiation.

The ABFT procedures to detect or correct errors are executed directly in the GPU, increasing the instruction caches requirements and scheduler strain. The higher number of operations needed to implement the ABFT procedure has the drawback of increasing the probability of having a DUE, which is in accordance with experimental results. Nevertheless, the DUE increase is not remarkable, being 12% for MxM and 14% for FFT, definitely lower than the 43% to 55% increment imposed by ECC.

ABFT correction capabilities were analytically calculated independently on input size (HUANG; ABRAHAM, 1984; JOU; ABRAHAM, 1988). It is then reasonable to extend the reported resilience results to other input sizes. When implemented on GPUs, the

complexity of ABFT procedures is linear with input data (PILLA et al., 2014; RECH et al., 2013a). The imposed ABFT overhead, both regarding performance and DUE caused by ABFT procedure corruption, is then expected to have a lower impact with increasing input size.

DWC strategies were also efficient, reducing the occurrence and even eliminating SDCs for $Hotspot$. As DWC design is independent of the code, the imposed overhead is also likely to be independent of the code, and so the detection capabilities. The results and discussion that derive from $Hotspot$ analysis can then be applied to other parallel codes without significant variations. All errors were corrected with *Spatial*, while in *E-O Spatial* few errors were left undetected as the redundant block can be executed in the same SM and a shared resource error propagates to both copies. DWC strategies reduce the SDC rate by one order of magnitude with respect to ECC.

The DUE rate of *Spatial* and *E-O Spatial* are comparable with the unhardened version and almost 50% lower than ECC-protected version. Only *Time* succeeds in reducing DUE occurrences significantly. In both, *Spatial* and *E-O Spatial*, the number of blocks to be dispatched is duplicated, increasing significantly the scheduler strain required for computation (which has been demonstrated to be particularly critical (RECH et al., 2014)). On the contrary, *Time* instantiates exactly the same number of blocks and threads of the unhardened version of $Hotspot$. Additionally, critical operations for a DUE, like blocks/threads dispatch, are interleaved with double operations in *Time*. This operation mix means that it is more likely for a neutron to generate a correctable SDC than a critical DUE for *Time*.

Still, DWC comes with a non-negligible computational overhead, which ranges from 90% for *Time* to 151% for *Spatial*. *Time* performs better as there is no scheduler overhead, while in *Spatial* and *E-O Spatial* more blocks need to be created and scheduled. DWC pays its generality and ease of implementation with poor performance. DWC offers high detection capabilities and may be particularly suitable for a safety-critical application. In HPC, DWC usage should be carefully engineered to avoid excessive performance degradation.

The combination of ECC and ABFT or ECC and DWC is likely to provide an even higher reliability. Nevertheless, the resulting error rate would be extremely low even with the accelerated particle beams used in this evaluation, preventing the observation of a statistically significant amount of failures.

Table 8.2: *DGEMM* CAROL-FI results.

Variable Name	File Name	Line Number	Overhead	PVF Contribution
block	dgemm.c	263	< 0.01%	0.28%
file2	dgemm.c	220	< 0.01%	0.07%
order	dgemm.c	262	< 0.01%	10.77%
A	dgemm.c	261	< 0.01%	1.48%
B	dgemm.c	261	< 0.01%	1.48%
file	dgemm.c	220	< 0.01%	0.21%
jj	dgemm.c	167	< 0.01%	1.04%
kk	dgemm.c	167	< 0.01%	3.87%
ii	dgemm.c	167	0.01%	5.11%
BB	dgemm.c	171	0.12%	4.94%
A	dgemm.c	219	0.12%	1.66%
B	dgemm.c	219	0.12%	1.21%
M	dgemm.c	240	0.12%	0.28%
i	dgemm.c	221	0.12%	2.59%
i	dgemm.c	242	0.12%	0.35%
j	dgemm.c	221	0.12%	1.28%
j	dgemm.c	242	0.12%	0.24%
j	dgemm.c	258	0.12%	0.07%
C	dgemm.c	261	0.12%	10.77%
jg	dgemm.c	167	0.26%	5.11%
k	dgemm.c	167	0.62%	2.90%
kg	dgemm.c	167	0.62%	5.76%
C	dgemm.c	165	4.54%	22.06%
j	dgemm.c	167	4.78%	2.76%
AA	dgemm.c	171	7.73%	0.10%
CC	dgemm.c	171	15.38%	4.49%
i	dgemm.c	167	32.42%	3.31%
ig	dgemm.c	167	32.42%	5.83%

8.2 Selective Hardening

Selective hardening, similar to Duplication With Comparison, can be implemented in several ways. Using redundant multithreading implemented in hardware, Mukherjee selectively hardened only a limited sphere of replication in (MUKHERJEE; KONTZ; REINHARDT, 2002). Moreover, at the register-instruction level, several works evaluate and implement selective hardening (CHIELLE et al., 2013; RESTREPO-CALLE et al., 2013; CHIELLE et al., 2015).

Since HPC devices cannot be modified, we focus on software-based hardening strategies. The fundamental point of selective hardening is the selection of what to harden.

A probabilistic and statistical model can be used to analyze the code, at assembly level for instance, and select some registers to be duplicated. The probabilistic model has the advantage of being very fast and uses few resources, but it does not take into account the impact of faults in the specific algorithm (i.e., faults can be masked).

Fault injection can also be used to analyze the critical portions of the code to be hardened. Using fault injection, we can more easily capture the impact of a fault in a specific resource. The downside is that pure fault injection does not take into account the probability of that resource to be affected by a fault. Another disadvantage is the time cost to evaluate all possible resources.

The approach used in this work is the fault injection one. CAROL-FI, described in section 6.1.1, was used to select the critical portions to harden. The disadvantages of the fault injection approach are mitigated due to the low CAROL-FI overhead and the methodology of randomly inject a fault at each execution, which reduces the probability to select a variable that is alive for a short period of the execution.

To evaluate the impact of selective hardening we use CAROL-FI on *DGEMM*, *LavaMD*, *HotSpot*, and *LUD*. We injected $69,657$ faults observing $5,956$ SDCs. The machine used during tests was an Intel Xeon Phi Knights Landing (KNL), which is the next generation of KNC described in section 5.1.2. The KNL has a total 68 physical cores, 34 MB of L2 cache and is built using $14nm$ technology. KNL is fully supported by CAROL-FI and is used by three of the current top10 supercomputers: Trinity at Los Alamos National Laboratory, Cori at Lawrence Berkeley National Laboratory, and Oakforest-PACS at Japan.

CAROL-FI results show how many of all the observed SDCs each variable produced. Then, we compute the code's PVF considering all SDCs, after we compute a new PVF without the SDCs observed in a variable. Thus, the difference between those two values is the PVF contribution of that specific variable. The PVF contribution shows us how much the code resilience would improve if we protect such variable by selective hardening.

To provide the best selective hardening for a code, we should take into account the overhead when selecting variables to harden. We want, for instance, to increase as much as possible the code resilience up to an acceptable overhead. In the HPC context, the execution time performance is usually the overhead of choice. Then, for example, one could accept a selective hardening with a maximum overhead of 10% the original execution time if this selective hardening reduces at least 50% of SDCs.

Table 8.3: *HotSpot* CAROL-FI results.

Variable Name	File Name	Line Number	Overhead	PVF Contribution
i	hotspot.c	189	< 0.01%	1.76%
i	hotspot.c	170	< 0.01%	6.12%
r	hotspot.c	168	< 0.01%	2.67%
t	hotspot.c	169	< 0.01%	2.67%
fp	hotspot.c	190	< 0.01%	1.37%
grid_cols	hotspot.c	249	< 0.01%	15.96%
grid_rows	hotspot.c	249	< 0.01%	22.93%
sim_time	hotspot.c	249	< 0.01%	7.23%
power	hotspot.c	250	< 0.01%	1.63%
temp	hotspot.c	250	< 0.01%	15.50%
num_omp_threads	hotspot.c	43	< 0.01%	6.84%
result	hotspot.c	250	< 0.01%	3.65%
j	hotspot.c	189	1.29%	1.43%
val	hotspot.c	214	2.58%	0.39%
vect	hotspot.c	209	2.58%	4.56%
i	hotspot.c	211	2.58%	1.43%
r	hotspot.c	54	16.85%	0.07%
result	hotspot.c	49	74.10%	3.78%

To measure the overhead a harden variable would introduce, we measure how many instructions write into that variable. The number of writes means that every time the variable is written, a consistency check is performed with a copy of that variable. The overhead metric can be changed depending on the hardening technique used. The number of writes is consistent with the Duplication With Comparison that needs to keep track of the variable changes and perform consistency checks.

Tables 8.2, 8.3, 8.4, and 8.5 show the PVF contribution and overhead of each variable that caused SDCs during fault injection campaign. The overhead column presents the percentage of writes considering only the variables that caused SDCs. Thus, 100% overhead means that we protected all critical variables only. The PVF contribution column is also a percentage, then, the sum of all critical variables is 100% of the original PVF. Repeated names in the first columns are possible because a variable with the same name can be declared and used in another scope. Thus, the second and third column show the filename and line number the variable is declared.

Finally, after measuring, or estimating, the overhead, one can prioritize the variables to harden by dividing the PVF contribution by the overhead. Then, we can select the variables with this prioritization up to a specific overhead limit. However, such prioritization does not guarantee the best variables selection for a specific limit. To get the best

Table 8.4: *LavaMD* CAROL-FI results.

Variable Name	File Name	Line Number	Overhead	PVF Contribution
file	main.c	112	< 0.01%	2.62%
dim_cpu	main.c	30	< 0.01%	2.04%
j	main.c	27	< 0.01%	0.87%
par_cpu	main.c	29	< 0.01%	4.37%
fv	kernel/kernelz_cpu.c	12	0.01%	9.62%
k	main.c	27	0.01%	0.58%
nh	main.c	36	0.01%	0.29%
m	main.c	27	0.11%	0.58%
qv	kernel/kernel_cpu.c	12	0.22%	17.20%
rv	kernel/kernel_cpu.c	12	0.23%	14.87%
n	main.c	27	0.34%	1.75%
qv_cpu	main.c	33	0.94%	7.00%
box_cpu	main.c	31	1.33%	0.58%
rv_cpu	main.c	32	3.78%	12.24%
i	main.c	27	4.72%	4.96%
fv_cpu	main.c	34	11.33%	16.03%
d	kernel/kernel_cpu.c	40	76.96%	4.37%

selection we need to solve the knapsack problem, but the prioritization can give a good approximation in a much shorter time.

8.2.1 Evaluation

Figures 8.4, 8.6, 8.5, and 8.7 present the efficiency of the selective hardening as we increase the number of variables protected for *DGEMM*, *HotSpot*, *LavaMD*, and *LUD* respectively. The X-axis shows the number of variables protected. The blue line presents the PVF coverage, and thus, when all variables are protected, the PVF coverage is 100%. The red line shows the overhead introduced ranging from 0%, without any hardening, up to 100% when all variables are protected. Thus, the higher the blue line is, the better. However, the red line is the opposite, and we want to keep it as low as possible. Furthermore, the overhead line does not increase linearly with the number of variables protected since each variable has its own cost of overhead.

DGEMM selective hardening is depicted in Figure 8.4. The C variable, which is the output array, is responsible for 22% of the overall PVF and almost 5% overhead (refer to Table 8.2). The C variable is the most critical one. However, we can achieve a similar PVF protection with nine variables selected by the prioritization of PVF contribu-

Table 8.5: *LUD* CAROL-FI results.

Variable Name	File Name	Line Number	Overhead	PVF Contribution
m	lud_main.c	50	< 0.01%	15.92%
matrix_dim	lud_main.c	45	< 0.01%	26.59%
chunks_in_inter_row	lud_omp.c	45	< 0.01%	2.20%
offset	lud_omp.c	45	< 0.01%	2.37%
temp	lud_omp.c	75	0.01%	0.25%
i_global	lud_omp.c	127	0.17%	2.79%
j_global	lud_omp.c	127	0.17%	4.06%
i	lud_omp.c	127	3.48%	2.79%
temp_left	lud_omp.c	129	3.52%	2.96%
temp_top	lud_omp.c	128	3.52%	5.84%
k	lud_omp.c	127	5.24%	1.10%
a	lud_omp.c	43	39.59%	27.52%
sum	lud_omp.c	130	44.31%	5.59%

tion divided by overhead. These nine variables provide a PVF coverage of 24% with an overhead inferior to 0.1%. Therefore, protecting variables based solely on PVF criticality can lead to a much worse PVF/overhead ratio than using the prioritization. Moreover, *DGEMM* achieves a PVF protection of 86% with an overhead of 12% when protecting 25 variables. After 25 variables, the overhead grows substantially higher than the PVF coverage.

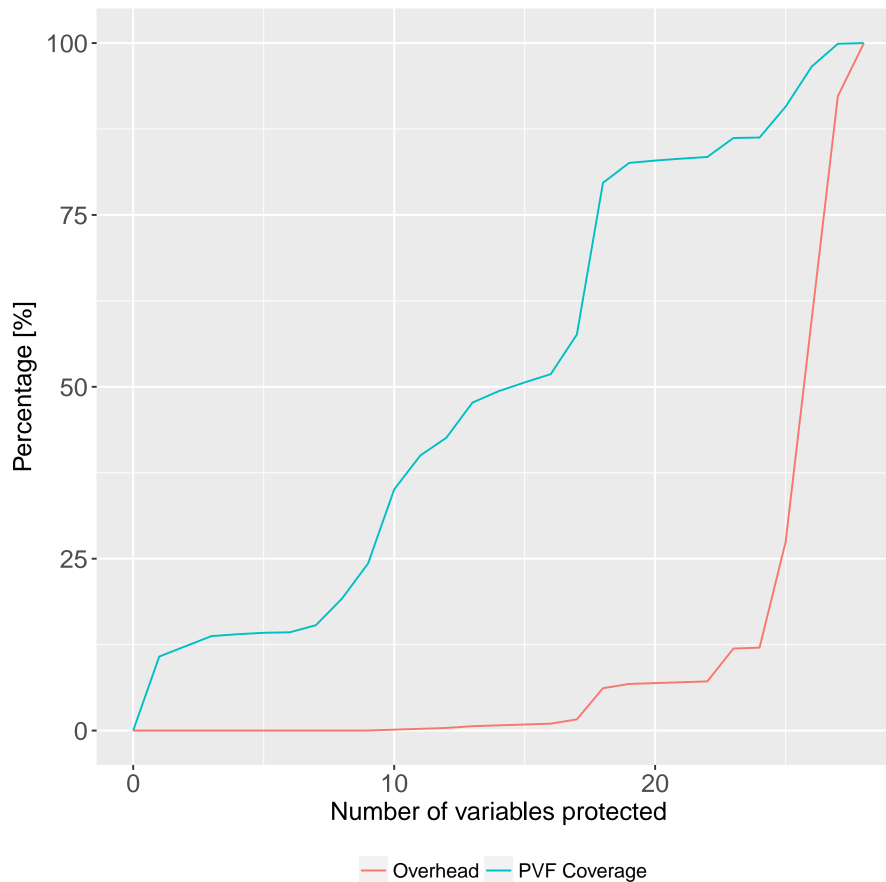
HotSpot selective hardening efficiency is shown in Figure 8.5. *HotSpot* shows a more drastic PVF coverage than *DGEMM*. *HotSpot* achieves a PVF coverage of 95% with an overhead lower than 10%. Moreover, we can reach a PVF coverage of 90% with an overhead lower than 1% when we protect twelve variables. This efficiency is possible because most critical variables are scalar variables that are rarely modified. Two variables are responsible for 90% of the overhead, but their PVF contribution is lower than 5% (refer to Table 8.3).

The *HotSpot* result is in line with the radiation criticality result we showed in section 6.2, *HotSpot* algorithm can naturally recover from most faults when converging to the solution. Thus, the higher overhead variables, such as the result matrix, which is constantly updated when converging to a solution, has a lower PVF contribution.

Figure 8.6 shows the selective hardening efficiency for *LavaMD*. To reach 90% of PVF coverage for *LavaMD*, one needs to accept an overhead of 15%. However, we can still achieve 60% PVF coverage with an overhead below 2% when protecting 12 variables.

LUD result is depicted in Figure 8.5. *LUD* shows the least promising result among

Figure 8.4: *DGEMM* selective hardening.



Source: The authors

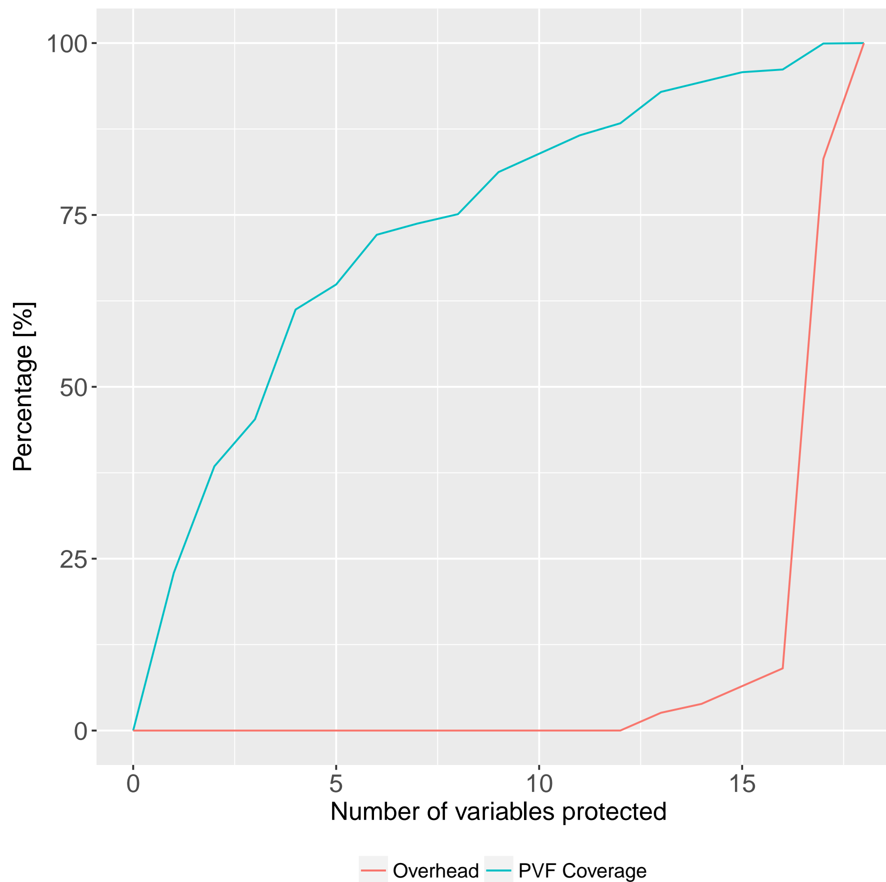
the four codes evaluated. However, we can still achieve 60% PVF coverage with an overhead of 3% protecting eight variables. The overhead would increase to about 50% or more to reach a PVF coverage higher than 65%. However, even if we limit the overhead to only 3%, we still reach a significant PVF coverage of 60% making it worth to implement selective hardening.

8.3 Discussion

We have compared the protection capabilities of different hardening techniques implemented in hardware and software. We showed that ECC provides the weakest SDC protection, but if the code is not memory-bound, like *FFT*, ECC provides the best overhead. Software-techniques specific to an algorithm still provide low overhead but guarantee stronger protection than ECC. A generic software-technique such as DWC provides complete protection against SDCs, but the overhead is unacceptable to HPC application.

To improve further the SDC protection for HPC applications, without the overhead

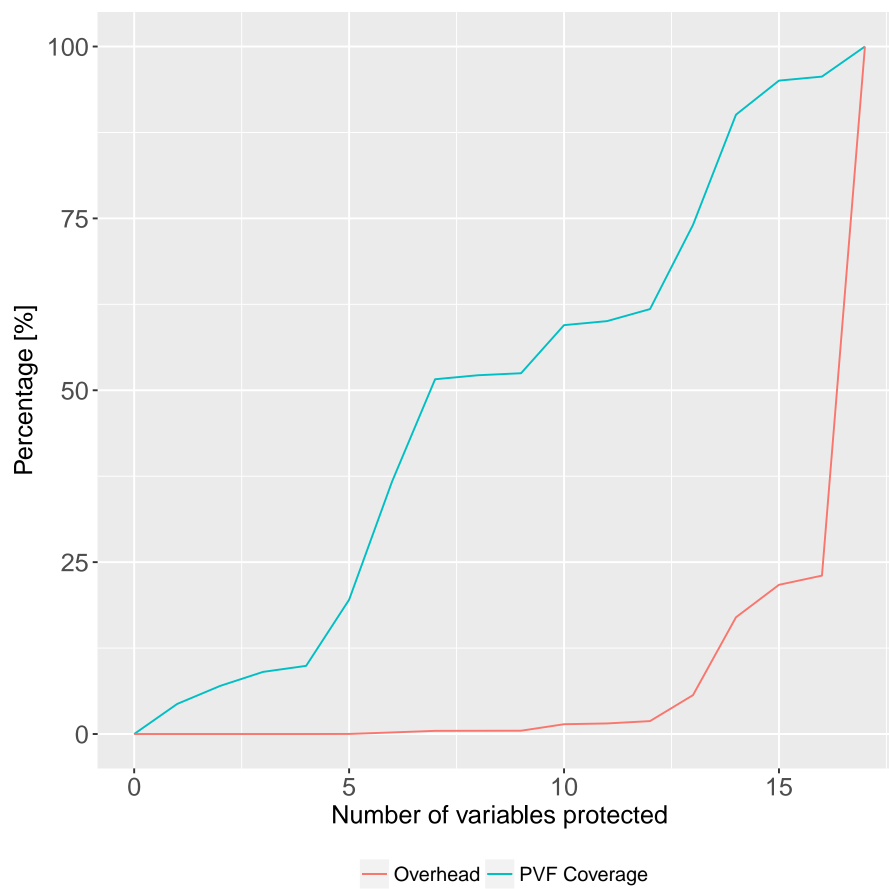
Figure 8.5: *HotSpot* selective hardening.



Source: The authors

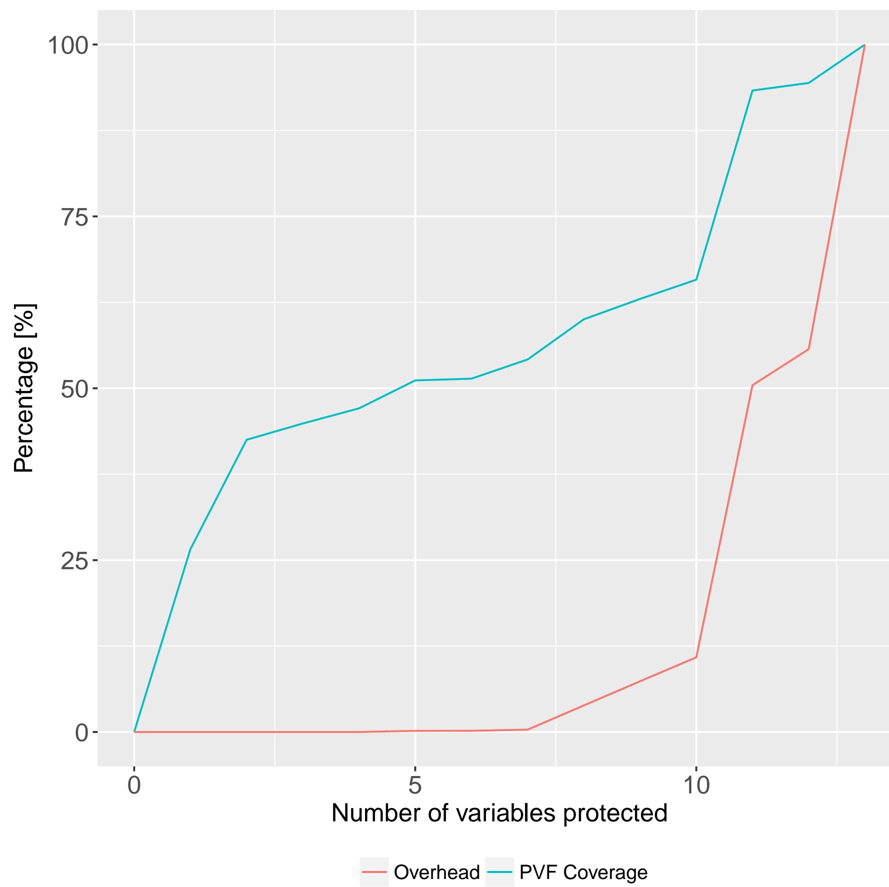
of a complete DWC, we introduced and evaluated the impact of a selective hardening. Thus, we evaluate using fault injection campaign the criticality of the code's variables. Then, we show that selective hardening can provide protection similar to a full DWC, with an overhead lower than specific techniques like ABFT. In general, the four codes tested achieved protection of 60% with an overhead below 3%.

Figure 8.6: *LavaMD* selective hardening.



Source: The authors

Figure 8.7: *LUD* selective hardening.



Source: The authors

9 CONCLUSION

This thesis presents an extensive study to *evaluate*, *understand*, and to provide *mitigation strategies* for the reliability issue in HPC applications. We present in-depth *evaluation* of GPU radiation sensitivity both at low-level – by accessing the raw memory structures error rate – and at operative-level – by measuring the Silent Data Corruption and Detectable Uncorrectable Error rate of a representative set of parallel applications. The presented data serves as a pragmatic and precise estimation of the realistic error rate of modern GPUs exposed to the natural neutron beam.

To *understand* better the issue, we go beyond the sole error rate and also evaluate how the errors spread, and the severity of SDCs for the two most prominent HPC devices used to accelerate performance in the supercomputers, NVIDIA GPUs and Intel Xeon Phis. 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 by accepting a certain error margin in the output.

We also 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).

To broaden even more our *understanding* of reliability issues, we have compared the reliability of an extensive set of devices and codes measured through both beam radiation experiments and, whenever available, fault-injection. The results data highlights both code-dependent and device-dependent behaviors. We found that the DUE rate is mostly device-dependent and cannot be easily predicted using fault-injection. This work also shows that efficient use of a higher amount of resources justifies the increase in FIT rate. Moreover, FinFET transistors are effective in reducing both the SDC and DUE rates independently of the code, while ECC is more effective in memory-bound codes. Softwares with long memory latencies have a higher SDC rate, especially when ECC is not available. Furthermore, additional resources, although increasing the FIT rates, generally improve MWBF. We have also qualified the observed errors regarding the difference between the expected value and the observed one, and the percentage of the output that has

been corrupted. Stencil applications, independently of the device, could take great advantage of approximate computing as 90% of corrupted executions could be accepted as correct with a 0.2% of output approximation. While KNCs spread errors because of memory coherency, GPUs corrupt fewer elements. Finally, ECC is shown to correct mainly errors that have a lower impact on the output.

Finally, to develop *mitigation strategies*, the available hardening solutions, including ECC, ABFT, and duplication with comparison, were analyzed and their efficiency and efficacy experimentally and analytically compared. We observed that ECC has the weakest protection with the best overhead while duplication has the strongest protection with the worst overhead. Then, using pragmatic information acquired using the homemade fault injection tool CAROL-FI, we demonstrate that selective hardening achieves 60% of the fault coverage with a low overhead similar to ECC. Thus, selective hardening can provide the best solution for HPC merging the efficacy of duplication with the efficiency of ECC.

9.1 Publications

In this section, we list the papers published as a result of the work performed during this thesis. We have published five papers in journals detailing the radiation experiments analysis and hardening strategies. The journal papers are listed in the following.

- FRATIN, V. et al. Energy-delay-fit product to compare processors and algorithm implementations. **Microelectronics Reliability**, v. 84, p. 112–120, May 2018. ISSN 0026-2714.
- LUNARDI, C. et al. Experimental and analytical analysis of sorting algorithms error criticality for hpc and large servers applications. **IEEE Transactions on Nuclear Science**, v. 64, n. 8, p. 2169–2178, Aug 2017. ISSN 0018-9499.
- OLIVEIRA, D. et al. Evaluation and mitigation of radiation-induced soft errors in graphics processing units. **IEEE Transactions on Computers**, v. 65, n. 3, p. 791–804, March 2016. ISSN 0018-9340.
- PILLA, L. et al. Memory access time and input size effects on parallel processors reliability. **IEEE Transactions on Nuclear Science**, v. 62, n. 6, p. 2627–2634, Dec 2015. ISSN 0018-9499.

- OLIVEIRA, D. et al. Modern gpus radiation sensitivity evaluation and mitigation through duplication with comparison. **IEEE Transactions on Nuclear Science**, v. 61, n. 6, p. 3115–3122, Dec 2014. ISSN 0018-9499.

We have also published nine papers in conferences and workshops. We published the CAROL-FI developed in the thesis context as well as the neutron beam and fault injection analysis presented in this thesis. The conference papers are listed in the following.

- FRATIN, V. et al. Code-dependent and architecture-dependent reliability behaviors. In: **to appear in 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks**. Luxembourg City, Luxembourg, 2018. ISSN 1530-0889.

- OLIVEIRA, D. et al. Experimental and analytical study of xeon phi reliability. In: **Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis**. New York, NY, USA: ACM, 2017. (SC '17), p. 28:1–28:12. ISBN 978-1-4503-5114-0.

- OLIVEIRA, D. et al. Radiation-induced error criticality in modern hpc parallel accelerators. In: **2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)**. Austin, TX, USA, 2017. p. 577–588.

- OLIVEIRA, D. et al. Carol-fi: An efficient fault-injection tool for vulnerability evaluation of modern hpc parallel accelerators. In: **Proceedings of the Computing Frontiers Conference**. New York, NY, USA: ACM, 2017. (CF'17), p. 295–298. ISBN 978-1-4503-4487-6.

- OLIVEIRA, D. et al. Input size effects on the radiation-sensitivity of modern parallel processors. In: **2016 IEEE Radiation Effects Data Workshop (REDW)**. Portland, OR, USA, 2016. p. 1–6.

- TIWARI, D. et al. Understanding GPU Errors on Large-scale HPC Systems and the Implications for System Design and Operation. In: **21st IEEE Symp. on High Performance Computer Architecture**. Burlingame, CA, USA, 2015. p. 331–342.

- OLIVEIRA, D. et al. The path to exascale: Code optimizations and hardening solutions reliability. In: **Proceedings of the 5th Workshop on Fault Tolerance for HPC at eXtreme Scale**. New York, NY, USA: ACM, 2015. (FTXS '15), p. 55–62. ISBN 978-1-4503-3569-0.

- OLIVEIRA, D. A. G. et al. Radiation Sensitivity of High Performance Computing Applications on Kepler-Based GPGPUs. In: **IEEE. International Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS 2014), co-located with IEEE International Conference on Dependable Systems and Networks (DSN 2014)**. Atlanta, USA, 2014. p. 732–737.

- OLIVEIRA, D. et al. GPGPUs ECC Efficiency and Efficacy. In: IEEE **International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems**. Amsterdam, Netherlands, 2014. p. 209–215.

REFERENCES

ALKHALIFA, Z. et al. Design and evaluation of system-level checks for on-line control flow error detection. **IEEE Transactions on Parallel and Distributed Systems**, v. 10, n. 6, p. 627–641, Jun 1999. ISSN 1045-9219.

ARGYRIDES, C. A. et al. Single element correction in sorting algorithms with minimum delay overhead. In: **2009 10th Latin American Test Workshop**. Buzios, Rio de Janeiro, Brazil: IEEE, 2009. p. 1–6. ISSN 2373-0862.

ASADI, G.-H. et al. Balancing Performance and Reliability in the Memory Hierarchy. In: **Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, 2005**. Washington, DC, USA: IEEE Computer Society, 2005. (ISPASS '05), p. 269–279. ISBN 0-7803-8965-4.

ASANOVIC, K. et al. **The Landscape of Parallel Computing Research: A View from Berkeley**. USA, 2006. Disponível em: <http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html>.

ATKINSON, B. et al. Fault injection experiments with the clamr hydrodynamics mini-app. In: **Software Reliability Engineering Workshops (ISSREW), 2014 IEEE International Symposium on**. Naples, Italy: IEEE, 2014. p. 6–9.

AUMANN, Y.; BENDER, M. A. Fault tolerant data structures. In: IEEE. **Foundations of Computer Science, 1996. Proceedings., 37th Annual Symposium on**. Burlington, VT, USA, 1996. p. 580–589.

BAILEY, D. et al. The NAS Parallel Benchmarks. **NASA Ames Research Center, RNR Technical Report RNR-94-007**, 1994. Disponível em: <http://hpc.sagepub.com/cgi/content/abstract/5/3/63>.

BAUMANN, R. Radiation-Induced Soft Errors in Advanced Semiconductor Technologies. **Device and Materials Reliability, IEEE Transactions on**, v. 5, n. 3, p. 305–316, Sept 2005. ISSN 1530-4388.

BAUMANN, R. C. Soft errors in commercial semiconductor technology: Overview and scaling trends. **IEEE 2002 Reliability Physics Tutorial Notes, Reliability Fundamentals**, v. 7, n. 1, 2002.

BAUTISTA-GOMEZ, L. et al. Fti: High performance fault tolerance interface for hybrid systems. In: **Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis**. New York, NY, USA: ACM, 2011. (SC '11), p. 1–12. ISBN 978-1-4503-0771-0. Disponível em: <http://doi.acm.org/10.1145/2063384.2063427>.

BERROCAL, E. et al. Exploring partial replication to improve lightweight silent data corruption detection for hpc applications. In: DUTOT, P.-F.; TRYSTRAM, D. (Ed.). **Euro-Par 2016: Parallel Processing: 22nd International Conference on Parallel and Distributed Computing, Grenoble, France, August 24-26, 2016, Proceedings**. Cham: Springer International Publishing, 2016. p. 419–430. ISBN 978-3-319-43659-3. Disponível em: <http://dx.doi.org/10.1007/978-3-319-43659-3_31>.

BIENIA, C. et al. The parsec benchmark suite: characterization and architectural implications. In: ACM. **Proceedings of the 17th international conference on Parallel architectures and compilation techniques**. Toronto, ON, Canada, 2008. p. 72–81.

BLUM, M. et al. Checking the correctness of memories. **Algorithmica**, Springer, v. 12, n. 2-3, p. 225–244, 1994.

BREUER, M. A. Multi-media applications and imprecise computation. In: IEEE. **Digital System Design, 2005. Proceedings. 8th Euromicro Conference on**. Porto, Portugal, 2005. p. 2–7.

BREUER, M. A.; GUPTA, S. K.; MAK, T. M. Defect and error tolerance in the presence of massive numbers of defects. **IEEE Design Test of Computers**, v. 21, n. 3, p. 216–227, May 2004. ISSN 0740-7475.

BRODAL, G. S.; JØRGENSEN, A. G.; MØLHAVE, T. Fault tolerant external memory algorithms. In: **International Symposium on Algorithms and Data Structures**. Berlin, Heidelberg: Springer, 2009. p. 411–422.

CHE, S. et al. Rodinia: A benchmark suite for heterogeneous computing. In: IEEE. **Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on**. 2009. p. 44–54. Disponível em: <http://dx.doi.org/10.1109/IISWC.2009.5306797>.

CHIELLE, E. et al. Improving error detection with selective redundancy in software-based techniques. In: IEEE. **2013 14th Latin American Test Workshop - LATW**. Cordoba, Argentina, 2013. p. 1–6. ISSN 2373-0862.

CHIELLE, E. et al. S-seta: Selective software-only error-detection technique using assertions. **IEEE Transactions on Nuclear Science**, v. 62, n. 6, p. 3088–3095, Dec 2015. ISSN 0018-9499.

CHRISTIANO, P.; DEMAINE, E. D.; KISHORE, S. Lossless fault-tolerant data structures with additive overhead. In: **International Symposium on Algorithms and Data Structures**. Berlin, Heidelberg: Springer, 2011. p. 243–254.

CHU, M.; KANNAN, S.; MCGREGOR, A. Checking and spot-checking the correctness of priority queues. In: **Automata, Languages and Programming**. Berlin, Heidelberg: Springer, 2007. p. 728–739.

CONSTANTINESCU, C. Impact of deep submicron technology on dependability of vlsi circuits. In: IEEE. **Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on**. Washington, DC, USA, 2002. p. 205–209.

CURRY, M. L. et al. Accelerating reed-solomon coding in raid systems with gpus. In: IEEE. **Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on**. Miami, FL, USA, 2008. p. 1–6. ISSN 1530-2075.

DANALIS, A. et al. The Scalable Heterogeneous Computing (SHOC) Benchmark Suite. In: **Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units**. New York, NY, USA: ACM, 2010. (GPGPU '10), p. 63–74. ISBN 978-1-60558-935-0.

DODD, P. et al. Neutron-induced soft errors, latchup, and comparison of ser test methods for sram technologies. In: IEEE. **Electron Devices Meeting, 2002. IEDM'02. International**. San Francisco, CA, USA, 2002. p. 333–336.

DONGARRA, J.; MEUER, H.; STROHMAIER, E. **TOP500 Supercomputer Sites: Nov. 2015**. 2015. Disponível em: <http://www.top500.org>.

EGWUTUOHA, I. P. et al. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. **The Journal of Supercomputing**, v. 65, n. 3, p. 1302–1326, 2013. ISSN 1573-0484. Disponível em: <http://dx.doi.org/10.1007/s11227-013-0884-0>.

ERNST, D. et al. Razor: circuit-level correction of timing errors for low-power operation. **IEEE Micro**, v. 24, n. 6, p. 10–20, Nov 2004. ISSN 0272-1732.

FANG, B. et al. Gpu-qin: A methodology for evaluating the error resilience of gpgpu applications. In: IEEE. **Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on**. Monterey, CA, USA, 2014. p. 221–230.

FANG, Y. P.; OATES, A. S. Characterization of single bit and multiple cell soft error events in planar and finfet srams. **IEEE Transactions on Device and Materials Reliability**, v. 16, n. 2, p. 132–137, June 2016. ISSN 1530-4388.

FARAZMAND, N.; UBAL, R.; KAELI, D. Statistical fault injection-based avf analysis of a gpu architecture. **Proceedings of SELSE**, v. 12, p. 1–6, 2012.

FERRARO-PETRILLO, U.; FINOCCHI, I.; ITALIANO, G. F. The price of resiliency: a case study on sorting with memory faults. **Algorithmica**, Springer, v. 53, n. 4, p. 597–620, 2009.

FINOCCHI, I.; ITALIANO, G. F. Sorting and searching in faulty memories. **Algorithmica**, Springer, v. 52, n. 3, p. 309–332, 2008.

FRATIN, V. et al. Code-dependent and architecture-dependent reliability behaviors. In: IEEE. **to appear in 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks**. Luxembourg City, Luxembourg, 2018. ISSN 1530-0889.

FRATIN, V. et al. Energy-delay-fit product to compare processors and algorithm implementations. **Microelectronics Reliability**, v. 84, p. 112–120, May 2018. ISSN 0026-2714.

FREIVALDS, R. Fast probabilistic algorithms. In: **Mathematical Foundations of Computer Science 1979**. Heidelberg,Berlin: Springer, 1979, (Lecture Notes in Computer Science, v. 74). p. 57–69. ISBN 978-3-540-09526-2.

GASIOT, G.; GIOT, D.; ROCHE, P. Alpha-induced multiple cell upsets in standard and radiation hardened srams manufactured in a 65 nm cmos technology. **Nuclear Science, IEEE Transactions on**, IEEE, v. 53, n. 6, p. 3479–3486, 2006.

GOLOUBEVA, O. et al. Soft-error detection using control flow assertions. In: IEEE. **Defect and Fault Tolerance in VLSI Systems, 2003. Proceedings. 18th IEEE International Symposium on**. Boston, MA, USA, 2003. p. 581–588. ISSN 1550-5774.

GOMEZ, L. A. B. et al. GPGPUs: How to Combine High Computational Power with High Reliability. In: IEEE. **2014 Design Automation and Test in Europe Conference and Exhibition**. Dresden, Germany, 2014. p. 1–9.

GUAN, Q. et al. Towards building resilient scientific applications: Resilience analysis on the impact of soft error and transient error tolerance with the clamr hydrodynamics mini-app. In: IEEE. **Cluster Computing (CLUSTER), 2015 IEEE International Conference on**. Chicago, IL, USA, 2015. p. 176–179.

HARI, S. K. S. et al. Sassifi: Evaluating resilience of gpu applications. In: IEEE. **Proceedings of the Workshop on Silicon Errors in Logic-System Effects (SELSE)**. Austin, TX, USA, 2015. p. 1–6.

HARI, S. K. S. et al. Sassifi: An architecture-level fault injection tool for gpu application resilience evaluation. **International Symposium on Performance Analysis of Systems and Software**, p. 249–258, Apr 2017.

HONG, S.; KIM, S. A low-cost mechanism exploiting narrow-width values for tolerating hard faults in alu. **IEEE Transactions on Computers**, v. 64, n. 9, p. 2433–2446, Sept 2015. ISSN 0018-9340.

HSUEH, M.-C.; TSAI, T. K.; IYER, R. K. Fault injection techniques and tools. **Computer**, IEEE, v. 30, n. 4, p. 75–82, 1997.

HUANG, K.-H.; ABRAHAM, J. Algorithm-Based Fault Tolerance for Matrix Operations. **Computers, IEEE Transactions on**, C-33, n. 6, p. 518–528, June 1984. ISSN 0018-9340.

INTEL. **Intel Xeon Phi Coprocessor System Software Developers Guide**. 2015. Disponível em: <https://software.intel.com/sites/default/files/managed/09/07/xeon-phi-coprocessor-system-software-developers-guide.pdf>.

INTEL. **An Overview of Programming for Intel Xeon processors and Intel Xeon Phi coprocessors**. 2015. Disponível em: <http://download.intel.com/newsroom/kits/xeon/phi/pdfs/overview-programming-intel-xeon-intel-xeon-phi-coprocessors.pdf>.

JEDEC. **Measurement and Reporting of Alpha Particle and Terrestrial Cosmic Ray-Induced Soft Errors in Semiconductor Devices**. USA, 2006. Disponível em: <https://www.jedec.org/standards-documents/docs/jesd-89a>.

JOU, J.-Y.; ABRAHAM, J. Fault-Tolerant FFT Networks. **Computers, IEEE Transactions on**, v. 37, n. 5, p. 548–561, 1988. ISSN 0018-9340.

KANAWATI, G. A.; KANAWATI, N. A.; ABRAHAM, J. A. Ferrari: A flexible software-based fault and error injection system. **IEEE Transactions on computers**, IEEE, v. 44, n. 2, p. 248–260, 1995.

KIM, S.-H. et al. A low power and highly reliable 400Mbps mobile DDR SDRAM with on-chip distributed ECC. In: IEEE. **Solid-State Circuits Conference, 2007. ASSCC ’07. IEEE Asian**. Jeju, South Korea, 2007. p. 34–37.

KIRK, D. B.; WEN-MEI, W. H. **Programming massively parallel processors: a hands-on approach**. Burlington, MA, USA: Morgan Kaufmann, 2012.

KRüGER, J.; WESTERMANN, R. Linear algebra operators for gpu implementation of numerical algorithms. In: **ACM SIGGRAPH 2003 Papers**. New York, NY, USA: ACM, 2003. (SIGGRAPH '03), p. 908–916. ISBN 1-58113-709-5. Disponível em: <http://doi.acm.org/10.1145/1201775.882363>.

LABEL, K. A. et al. In: IEEE. **Aerospace Applications Conference, 1996. Proceedings., 1996 IEEE**. Aspen, CO, USA, 1996. v. 1, p. 375–390.

LAPRIE, J. C. Dependable computing and fault tolerance : Concepts and terminology. In: IEEE. **Fault-Tolerant Computing, 1995, Highlights from Twenty-Five Years., Twenty-Fifth International Symposium on**. Pasadena, CA, USA, 1995. p. 2–11.

LEE, V. W. et al. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. **SIGARCH Comput. Archit. News**, ACM, New York, NY, USA, v. 38, n. 3, p. 451–460, june 2010.

LUCAS, R. **Top Ten Exascale Research Challenges**. USA, 2014. Disponível em: <https://science.energy.gov/~/media/ascr/ascac/pdf/meetings/20140210/Top10reportFEB14.pdf>.

LUNARDI, C. et al. Experimental and analytical analysis of sorting algorithms error criticality for hpc and large servers applications. **IEEE Transactions on Nuclear Science**, v. 64, n. 8, p. 2169–2178, Aug 2017. ISSN 0018-9499.

LUTZ, R. R. Analyzing software requirements errors in safety-critical, embedded systems. In: IEEE. **Requirements Engineering, 1993., Proceedings of IEEE International Symposium on**. San Diego, CA, USA, 1993. p. 126–133.

MADEIRA, H.; SILVA, J. G. On-line signature learning and checking. In: **Dependable Computing for Critical Applications 2**. Berlin, Heidelberg: Springer, 1992. p. 395–420.

MAITRE, O. Understanding NVIDIA GPGPU Hardware. In: **Massively Parallel Evolutionary Computation on GPGPUs**. Berlin, Heidelberg: Springer, 2013. p. 15–34.

MARUYAMA, N.; NUKADA, A.; MATSUOKA, S. A high-performance fault-tolerant software framework for memory on commodity gpus. In: IEEE. **Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on**. Atlanta, GA, USA, 2010. p. 1–12. ISSN 1530-2075.

MEIXNER, A.; SORIN, D. J. Detouring: Translating software to circumvent hard faults in simple cores. In: IEEE. **2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)**. Anchorage, AK, USA, 2008. p. 80–89. ISSN 1530-0889.

MICHALAK, S. E. et al. Predicting the number of fatal soft errors in los alamos national laboratory's asc q supercomputer. **Device and Materials Reliability, IEEE Transactions on**, IEEE, v. 5, n. 3, p. 329–335, 2005.

MIREMADI, G. et al. Two software techniques for on-line error detection. In: IEEE. **Fault-Tolerant Computing, 1992. FTCS-22. Digest of Papers., Twenty-Second International Symposium on**. Boston, MA, USA, 1992. p. 328–335.

MITRA, S. System-level single-event effects. **Nuclear Science Short Course, IEEE Transactions on**, IEEE, 2012.

MOON, T. K. **Error correction coding: Mathematical Methods and Algorithms**. Hoboken, NJ, USA: Jhon Wiley and Son, 2005. ISBN 9780471739210.

MUKHERJEE, S. S.; KONTZ, M.; REINHARDT, S. K. Detailed design and evaluation of redundant multi-threading alternatives. In: IEEE. **Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on**. Anchorage, AK, USA, 2002. p. 99–110. ISSN 1063-6897.

MUKHERJEE, S. S. 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**. Washington, DC, USA: IEEE Computer Society, 2003. p. 29–40. ISBN 0-7695-2043-X.

NAKSINEHABOON, N. et al. Reliability-aware approach: An incremental checkpoint/restart model in hpc environments. In: IEEE. **Cluster Computing and the Grid, 2008. CCGRID '08. 8th IEEE International Symposium on**. Lyon, France, 2008. p. 783–788.

NEEDLEMAN S.B., W. C. A general method applicable to the search for similarities in the amino acid sequence of two proteins. **Journal of Molecular Biology**, v. 48, n. 2, p. 443–453, 1969.

NEUMANN, J. V. Probabilistic logics and the synthesis of reliable organisms from unreliable components. **Automata studies**, v. 34, p. 43–98, 1956.

NICOLAIDIS, M. Time redundancy based soft-error tolerance to rescue nanometer technologies. In: IEEE. **VLSI Test Symposium, 1999. Proceedings. 17th IEEE**. Washington, DC, USA, 1999. p. 86–94. ISSN 1093-0167.

NOH, J. et al. Study of neutron soft error rate (ser) sensitivity: Investigation of upset mechanisms by comparative simulation of finfet and planar mosfet srams. **Nuclear Science, IEEE Transactions on**, v. 62, n. 4, p. 1642–1649, Aug 2015. ISSN 0018-9499.

NVIDIA. **NVIDIA Kepler K20 GPU Datasheet**. 2012.

NVIDIA. **CUBLAS Library User Guide**. 2014. <http://docs.nvidia.com/cuda/pdf/CUBLAS_Library.pdf>.

NVIDIA. **CUDA C Programming Guide**. 2015. <http://docs.nvidia.com/cuda/cuda-c-programming-guide/>.

NVIDIA. **Kepler Tuning Guide :: CUDA Toolkit Documentation**. 2015. <http://docs.nvidia.com/cuda/kepler-tuning-guide/>.

NVIDIA. **NVIDIAs Next Generation CUDA Compute Architecture: Kepler GK110**. 2015. Disponível em: <http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf>.

NVIDIA. **Tesla K20 GPU Accelerator, Board Specification**. 2015. Disponível em: <http://www.nvidia.com/content/PDF/kepler/Tesla-K20-Passive-BD-06455-001-v07.pdf>.

NVIDIA. **NVIDIA TITAN X**. 2016. Https://www.nvidia.com/en-us/geforce/products/10series/titan-x-pascal/.

OH, N.; SHIRVANI, P. P.; MCCLUSKEY, E. J. Control-flow checking by software signatures. **IEEE Transactions on Reliability**, v. 51, n. 1, p. 111–122, Mar 2002. ISSN 0018-9529.

OH, N.; SHIRVANI, P. P.; MCCLUSKEY, E. J. Error detection by duplicated instructions in super-scalar processors. **IEEE Transactions on Reliability**, v. 51, n. 1, p. 63–75, Mar 2002. ISSN 0018-9529.

OHLSSON, J.; RIMEN, M. Implicit signature checking. In: IEEE. **Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on**. Pasadena, CA, USA, 1995. p. 218–227.

OLIVEIRA, D. **CAROL-FI Fault Injector**. Porto Alegre, RS, Brazil: GitHub, 2017. <https://github.com/UFRGS-CAROL/carol-fi>.

OLIVEIRA, D. et al. Carol-fi: An efficient fault-injection tool for vulnerability evaluation of modern hpc parallel accelerators. In: **Proceedings of the Computing Frontiers Conference**. New York, NY, USA: ACM, 2017. (CF'17), p. 295–298. ISBN 978-1-4503-4487-6.

OLIVEIRA, D. et al. Experimental and analytical study of xeon phi reliability. In: **Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis**. New York, NY, USA: ACM, 2017. (SC '17), p. 28:1–28:12. ISBN 978-1-4503-5114-0.

OLIVEIRA, D. et al. Input size effects on the radiation-sensitivity of modern parallel processors. In: IEEE. **2016 IEEE Radiation Effects Data Workshop (REDW)**. Portland, OR, USA, 2016. p. 1–6.

OLIVEIRA, D. A. G. et al. Radiation Sensitivity of High Performance Computing Applications on Kepler-Based GPGPUs. In: IEEE. **International Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS 2014), co-located with IEEE International Conference on Dependable Systems and Networks (DSN 2014)**. Atlanta, USA, 2014. p. 732–737.

OLIVEIRA, D. A. G. et al. GPGPUs ECC Efficiency and Efficacy. In: IEEE. **International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems**. Amsterdam, Netherlands, 2014. p. 209–215.

OLIVEIRA, D. A. G. et al. Modern gpus radiation sensitivity evaluation and mitigation through duplication with comparison. **IEEE Transactions on Nuclear Science**, v. 61, n. 6, p. 3115–3122, Dec 2014. ISSN 0018-9499.

OLIVEIRA, D. A. G. D. et al. Radiation-induced error criticality in modern hpc parallel accelerators. In: IEEE. **2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)**. Austin, TX, USA, 2017. p. 577–588.

OLIVEIRA, D. A. G. de et al. The path to exascale: Code optimizations and hardening solutions reliability. In: **Proceedings of the 5th Workshop on Fault Tolerance for**

HPC at eXtreme Scale. New York, NY, USA: ACM, 2015. (FTXS '15), p. 55–62. ISBN 978-1-4503-3569-0.

OLIVEIRA, D. A. G. de et al. Evaluation and mitigation of radiation-induced soft errors in graphics processing units. **IEEE Transactions on Computers**, v. 65, n. 3, p. 791–804, March 2016. ISSN 0018-9340.

OWENS, J. et al. GPU Computing. **Proceedings of the IEEE**, v. 96, n. 5, p. 879–899, 2008. ISSN 0018-9219.

PARASHAR, A.; SIVASUBRAMANIAM, A.; GURUMURTHI, S. Slick: Slice-based locality exploitation for efficient redundant multithreading. **SIGOPS Oper. Syst. Rev.**, ACM, New York, NY, USA, v. 40, n. 5, p. 95–105, out. 2006. ISSN 0163-5980. Disponível em: <http://doi.acm.org/10.1145/1168917.1168870>.

PASSERAT-PALMBACH, J. et al. **Warp-level parallelism: Enabling multiple replications in parallel on GPU**. 2015. <https://arxiv.org/abs/1501.01405>.

PELC, A. Searching games with errors–fifty years of coping with liars. **Theoretical Computer Science**, Elsevier, v. 270, n. 1, p. 71–109, 2002.

PILLA, L. et al. Software-based hardening strategies for neutron sensitive fft algorithms on gpus. **Nuclear Science, IEEE Transactions on**, v. 61, n. 4, p. 1874–1880, Aug 2014. ISSN 0018-9499.

PILLA, L. L. et al. Memory access time and input size effects on parallel processors reliability. **IEEE Transactions on Nuclear Science**, v. 62, n. 6, p. 2627–2634, Dec 2015. ISSN 0018-9499.

POWELL, M. D. et al. Architectural core salvaging in a multi-core processor for hard-error tolerance. In: **Proceedings of the 36th Annual International Symposium on Computer Architecture**. New York, NY, USA: ACM, 2009. (ISCA '09), p. 93–104. ISBN 978-1-60558-526-0. Disponível em: <http://doi.acm.org/10.1145/1555754.1555769>.

PUENTE, J. de la et al. Mimetic seismic wave modeling including topography on deformed staggered grids. **GEOPHYSICS**, v. 79, n. 3, p. T125–T141, 2014.

QUINN, H. et al. Using benchmarks for radiation testing of microprocessors and fpgas. **IEEE Transactions on Nuclear Science**, v. 62, n. 6, p. 2547–2554, Dec 2015. ISSN 0018-9499.

RECH, P. et al. An Efficient and Experimentally Tuned Software-Based Hardening Strategy for Matrix Multiplication on GPUs. **Nuclear Science, IEEE Transactions on**, v. 60, n. 4, p. 2797–2804, 2013. ISSN 0018-9499.

RECH, P. et al. Threads distribution effects on graphics processing units neutron sensitivity. **Nuclear Science, IEEE Transactions on**, v. 60, n. 6, p. 4220–4225, Dec 2013. ISSN 0018-9499.

RECH, P. et al. **UFRGS-CAROL**. Porto Alegre, RS, Brazil: GitHub, 2016. <https://github.com/UFRGS-CAROL/itc2016-log-data>.

RECH, P. et al. Impact of GPUs Parallelism Management on Safety-Critical and HPC Applications Reliability. In: IEEE. **IEEE International Conference on Dependable Systems and Networks (DSN 2014)**. Atlanta, USA, 2014. p. 455–466.

REIS, G. et al. Design and evaluation of hybrid fault-detection systems. In: **Proceedings of the 2005 International Symposium on Computer Architecture, ISCA'05**. Madison, WI, USA: IEEE Press, 2005. p. 148–159.

REIS, G. A. et al. Swift: software implemented fault tolerance. In: IEEE. **International Symposium on Code Generation and Optimization**. New York, NY, USA, 2005. p. 243–254.

RESEARCH, A. S. C. **Scientific Discovery through Advanced Computing - The Challenges of Exascale**. 2016. <https://science.energy.gov/ascr/research/scidac/exascale-challenges/>.

RESTREPO-CALLE, F. et al. Selective swift-r. **Journal of Electronic Testing**, Springer, v. 29, n. 6, p. 825–838, 2013.

SAGGESE, G. P. et al. An experimental study of soft errors in microprocessors. **IEEE Micro**, v. 25, n. 6, p. 30–39, Nov 2005. ISSN 0272-1732.

SANTINI, T. et al. Reliability analysis of operating systems and software stack for embedded systems. **IEEE Transactions on Nuclear Science**, v. 63, n. 4, p. 2225–2232, Aug 2016. ISSN 0018-9499.

SCHROEDER, B.; PINHEIRO, E.; WEBER, W.-D. Dram errors in the wild: a large-scale field study. **Communications of the ACM**, ACM, Seattle, WA, USA, v. 54, n. 2, p. 100–107, 2011.

SIMONITE, T. **Why a Chip That's Bad at Math Can Help Computers Tackle Harder Problems**. 2016. <https://www.technologyreview.com/s/601263/why-a-chip-thats-bad-at-math-can-help-computers-tackle-harder-problems/>.

SNIR, M. et al. Addressing failures in exascale computing. **International Journal of High Performance Computing Applications**, SAGE Publications, v. 28, n. 2, p. 129–173, 2014.

SRIDHARAN, V.; KAELI, D. R. Eliminating microarchitectural dependency from architectural vulnerability. In: IEEE. **2009 IEEE 15th International Symposium on High Performance Computer Architecture**. Raleigh, NC, USA, 2009. p. 117–128. ISSN 1530-0897.

TAN, J. et al. Analyzing soft-error vulnerability on GPGPU microarchitecture. In: IEEE. **Workload Characterization (IISWC), 2011 IEEE International Symposium on**. Austin, TX, USA, 2011. p. 226–235.

TIWARI, D. et al. Understanding GPU Errors on Large-scale HPC Systems and the Implications for System Design and Operation. In: IEEE. **21st IEEE Symp. on High Performance Computer Architecture**. Burlingame, CA, USA, 2015. p. 331–342.

TSAI, T. K.; IYER, R. K. Measuring fault tolerance with the ftape fault injection tool. In: SPRINGER. **International Conference on Modelling Techniques and Tools for Computer Performance Evaluation**. Berlin, Heidelberg, 1995. p. 26–40.

VIOLANTE, M. et al. 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**, v. 54, n. 4, p. 1184–1189, 2007. ISSN 0018-9499.

VOLKOV, V.; DEMMEL, J. W. Benchmarking GPUs to tune dense linear algebra. In: IEEE. **Proceedings of the 2008 ACM/IEEE conference on Supercomputing**. Austin, TX, USA, 2008. p. 1–11.

WANG, F. et al. Dependability analysis of nano-scale finfet circuits. In: IEEE. **IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures (ISVLSI'06)**. Karlsruhe, Germany, 2006. p. 6–pp. ISSN 2159-3469.

WEAVER, C. et al. Techniques to reduce the soft error rate of a high-performance microprocessor. In: **International Symposium on Computer Architecture (ISCA'04)**. Munchen, Germany: IEEE Press, 2004. p. 264–275.

WICKER, S. B. **Reed-Solomon Codes and Their Applications**. Piscataway, NJ, USA: IEEE Press, 1994. ISBN 078031025X.

WILKEN, K.; SHEN, J. P. Concurrent error detection using signature monitoring and encryption. In: SPRINGER. **Dependable Computing for Critical Applications**. Berlin, Heidelberg, 1991. p. 365–384.

WILKENING, M. et al. Calculating Architectural Vulnerability Factors for Spatial Multi-bit Transient Faults. In: IEEE. **Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture**. Cambridge, UK, 2014. p. 293–305.

WITTENBRINK, C. M.; KILGARIFF, E.; PRABHU, A. Fermi GF100 GPU architecture. **IEEE Micro**, IEEE, v. 31, n. 2, p. 50–59, 2011.

WOO, S. C. et al. The splash-2 programs: Characterization and methodological considerations. In: ACM. **ACM SIGARCH Computer Architecture News**. Santa Margherita Ligure, Italy, 1995. v. 23, n. 2, p. 24–36.

WUNDERLICH, H.-J.; BRAUN, C.; HALDER, S. Efficacy and efficiency of algorithm-based fault-tolerance on gpus. In: IEEE. **On-Line Testing Symposium (IOLTS), 2013 IEEE 19th International**. Chania, Greece, 2013. p. 240–243.

XILINX. **Zynq-7000 All Programmable SoC Technical Reference Manual UG585 (v1.11)**. 2016. Https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf.

YAU, S. S.; CHEN, F.-C. An approach to concurrent control flow checking. **IEEE Transactions on Software Engineering**, SE-6, n. 2, p. 126–137, March 1980. ISSN 0098-5589.

ZHANG, Y. et al. Runtime asynchronous fault tolerance via speculation. In: ACM. **Proceedings of the Tenth International Symposium on Code Generation and Optimization**. San Jose, CA, USA, 2012. p. 145–154.

ZIEGLER, J. F. et al. Ibm experiments in soft fails in computer electronics (1978–1994). **IBM journal of research and development**, IBM, v. 40, n. 1, p. 3–18, 1996.

ZIEGLER, J. F.; PUCHNER, H. **SER–history, Trends and Challenges: A Guide for Designing with Memory ICs**. San Jose, CA, USA: Cypress, 2004.