William & Mary
CHARTERED 1693

W&M ScholarWorks

Dissertations, Theses, and Masters Projects | Theses, Dissertations, & Master Projects

2019

GPGPU Reliability Analysis: From Applications to Large Scale Systems

Bin Nie
William & Mary - Arts & Sciences, bnie0307@gmail.com

Follow this and additional works at: https://scholarworks.wm.edu/etd

Part of the Computer Sciences Commons

Recommended Citation

Nie, Bin, "GPGPU Reliability Analysis: From Applications to Large Scale Systems" (2019). *Dissertations, Theses, and Masters Projects.* William & Mary. Paper 1563898932.
http://dx.doi.org/10.21220/s2-j086-2347

This Dissertation is brought to you for free and open access by the Theses, Dissertations, & Master Projects at W&M ScholarWorks. It has been accepted for inclusion in Dissertations, Theses, and Masters Projects by an authorized administrator of W&M ScholarWorks. For more information, please contact scholarworks@wm.edu.

GPGPU Reliability Analysis:
from Applications to Large Scale Systems

Bin Nie

Dalian, Liaoning, China

Master of Science, Fordham University, 2014
Bachelor of Engineer, Xiamen University, 2012

A Dissertation presented to the Graduate Faculty
of The College of William & Mary in Candidacy for the Degree of
Doctor of Philosophy

Department of Computer Science

College of William & Mary
May, 2019

© Copyright by Bin Nie 2019

APPROVAL PAGE

This Dissertation is submitted in partial fulfillment of
the requirements for the degree of

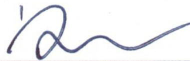
Doctor of Philosophy

Bin Nie

Approved by the Committee, May 2019

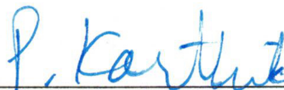
Committee Chair
Professor Evgenia Smirni, Computer Science
College of William & Mary

Professor Adwait Jog, Computer Science
College of William & Mary



Professor Xu Liu, Computer Science
College of William & Mary

Professor Bin Ren, Computer Science
College of William & Mary



Professor Karthik Pattabiraman, Electrical and Computer Engineering
University of British Columbia

ABSTRACT

Over the past decade, GPUs have become an integral part of mainstream high-performance computing (HPC) facilities. Since applications running on HPC systems are usually long-running, any error or failure could result in significant loss in scientific productivity and system resources. Even worse, since HPC systems face severe resilience challenges as progressing towards exascale computing, it is imperative to develop a better understanding of the reliability of GPUs. This dissertation fills this gap by providing an understanding of the effects of soft errors on the entire system and on specific applications.

To understand system-level reliability, a large-scale study on GPU soft errors in the field is conducted. The occurrences of GPU soft errors are linked to several temporal and spatial features, such as specific workloads, node location, temperature, and power consumption. Further, machine learning models are proposed to predict error occurrences on GPU nodes so as to proactively and dynamically turning on/off the costly error protection mechanisms based on prediction results.

To understand the effects of soft errors at the application level, an effective fault-injection framework is designed aiming to understand the reliability and resilience characteristics of GPGPU applications. This framework is effective in terms of reducing the tremendous number of fault injection locations to a manageable size while still preserving remarkable accuracy. This framework is validated with both single-bit and multi-bit fault models for various GPGPU benchmarks. Lastly, taking advantage of the proposed fault-injection framework, this dissertation develops a hierarchical approach to understanding the error resilience characteristics of GPGPU applications at kernel, CTA, and warp levels. In addition, given that some corrupted application outputs due to soft errors may be acceptable, we present a use case to show how to enable low-overhead yet reliable GPU computing for GPGPU applications.

TABLE OF CONTENTS

Acknowledgments vii

Dedication viii

List of Tables ix

List of Figures xi

1 Introduction 1

1.1 Contributions 5

1.1.1 GPU Reliability Analysis at the System Level 6

1.1.2 GPU Reliability Analysis at the Application Level 7

1.2 Organization 8

2 Background and Related Work 9

2.1 Benefits of Accelerators on Large-Scale Systems 9

2.2 General-Purpose GPUs for Scientific Computing 11

2.2.1 Baseline GPU Architecture 12

2.2.2 GPGPU Applications and Execution Model 13

2.2.3 GPU Errors in the Field 13

2.3 Reliability Analysis in the Field 14

2.3.1 System-Level Reliability Analysis 15

2.3.2 Application-Level Reliability Analysis 17

2.4 Chapter Summary 18

3 A Large-Scale Study of Soft-Errors on GPUs in the Field 19
3.1 Related Work 20
3.2 Methodology 22
3.2.1 Titan Supercomputer Organization and NVIDIA K20X GPU Architecture 22
3.2.2 GPU Errors: Collection and Analysis Methodology 24
3.2.3 Limitations and Scope 25
3.3 Analyzing Single Bit Errors (SBEs) 26
3.4 Analyzing Dynamic Page Retirement (DPR) Errors on the Titan Supercomputer 33
3.5 Analyzing Performance Variance in SBE and DPR Affected GPU Nodes 38
3.6 Effect of Temperature on Dynamic Page Retirement Errors 40
3.7 Chapter Summary 46

4 Characterization of Single-Bit Error in the Wild 47
4.1 Temperature Characteristics 49
4.2 GPU Error Characterization 53
4.2.1 SBE Offender Nodes 53
4.2.2 Application 54
4.2.3 Temperature and Power Consumption 55
4.2.3.1 A bird's eye view 56
4.2.3.2 Considering the time dimension 57
4.2.3.3 Considering the space dimension 60
4.3 Chapter Summary 61

5 Predicting GPU Soft-Errors with Neural Networks 63
5.1 Related Work 65

5.1.1 Applications of Machine Learning Models in Systems 65
5.1.2 Time Series Prediction 66
5.2 Overview of the Methodology 66
5.3 SBE Prediction Framework 69
5.3.1 Feature Selection 69
5.3.2 Challenge: Imbalanced Data Set 70
5.3.3 Model Selection 73
5.4 Evaluation 73
5.4.1 Evaluation with Oracle Data 74
5.4.2 PRACTISE for Feature Prediction 77
5.4.3 SBE Prediction with PRACTISE 78
5.5 Discussion 80
5.5.1 Application of SBE Prediction. 80
5.5.2 Open Problems and Challenges 81
5.6 Chapter Summary 81

6 Predicting GPU Soft-Errors with a Variety of Machine Learning Models 83
6.1 Feature Selection 84
6.1.1 Temporal Features 84
6.1.2 Spatial Features 85
6.2 Machine Learning Framework and Model 86
6.2.1 Overview 86
6.2.2 Challenge: Imbalanced Dataset 87
6.2.3 Two-Stage Machine Learning Models 88
6.2.4 Machine Learning Model Selection 90
6.3 Evaluation and Analysis 91
6.3.1 Data Description and Evaluation Metrics 91

6.3.2 Machine Learning Model Comparison 92
6.3.3 Feature Analysis 94
6.3.4 Prediction Analysis 97
6.4 Chapter Summary 99

7 Fault Site Pruning for Practical Reliability Analysis of GPGPU Applications 101
7.1 Related Work 105
7.2 Background and Methodology 107
7.2.1 Baseline Fault Injection Methodology 107
7.2.2 Baseline Fault Model 108
7.2.3 Statistical Considerations 108
7.3 Progressive Fault Site Pruning 110
7.3.1 Overview 110
7.3.2 Thread-Wise Pruning 112
7.3.2.1 CTA-wise Pruning 113
7.3.2.2 Thread-wise Pruning 116
7.3.3 Instruction-Wise Pruning 119
7.3.4 Loop-Wise Pruning 122
7.3.5 Bit-Wise Pruning 124
7.4 Evaluation 126
7.5 Multi-Bit Fault Injection 130
7.5.1 Assumptions 130
7.5.2 Extending Pruning to Multi-bit Fault Injection 131
7.5.2.1 Extending to double-bit fault injection 132
7.5.2.2 Extending to multi-bit fault injection 132
7.5.3 Evaluation 133
7.5.3.1 Comparison of accuracy 133

7.5.3.2 Impact of multi-bit faults 136
7.6 Chapter Summary 138

8 A Hierarchical Approach to Enabling Low-Overhead Reliable GPU Computing 139
8.1 Related Work 143
8.2 Evaluation Methodology 145
8.2.1 Benchmarks and Evaluation Metrics 145
8.2.2 Evaluation Framework 146
8.3 A Hierarchical Approach to Thread Classification 149
8.3.1 Multi-level Classification and Thread Selection 150
8.3.1.1 CTA-level classification 150
8.3.1.2 Warp-level classification 152
8.3.1.3 Classification result and thread selection 153
8.4 Hierarchical Approach to Error Resilience Characterization 156
8.4.1 Application Kernel Level Characteristics 156
8.4.1.1 Scope of accuracy-aware resilience 156
8.4.1.2 Sensitivity to input size 158
8.4.2 CTA Level Characteristics 159
8.4.3 Warp Level Characteristics 160
8.4.4 Statistical Validation 161
8.5 Use Case: Reducing Protection Overhead 162
8.6 Chapter Summary 167

9 Future Work 168
9.1 Fault Injection for Multiple Inputs 168
9.2 Low-Overhead Reliable GPU Computing 169
9.2.1 Thread-to-CTA Remapping 169

9.2.2 Resilience-Aware Scheduling 171

ACKNOWLEDGMENTS

I would like to thank everyone who has helped with this thesis, especially:

My advisor, Professor Evgenia Smirni, who encourages me all the time, continuously teaches me new knowledge, and brings lots of chances to me to grow up in the research world. Without her patient guidance and persistent help, this thesis would not be possible.

My committee members, Professor Adwait Jog, Professor Xu Liu, Professor Bin Ren, and Professor Karthik Pattabiraman for their great support and insightful feedback and comments.

My intern mentors, Dr. Devesh Tiwari in Northeastern University, Dr. Mehran Kafai and Dr. Kave Eshghi in Hewlett Packard Enterprise, and Dr. Jianwu Xu and Dr. Hui Zhang in NEC Labs America. Without their guidance and help, I would not go through so many interesting and exciting projects.

All the faculty and staff at the Computer Science Department of the College of William and Mary. Special thanks to Vanessa Godwin, Jacqulyn Johnson, and Dale Hayes for their considerate and efficient assistance.

Ji Xue, Lishan Yang, Jacob Alter and many other dear friends for their support and help.

Finally, and foremost, I would like to give special thanks to my husband, Zhuo, for his understanding and encouragement, and my parents for their support throughout this research endeavor.

I would like to dedicate this dissertation to my family, who provided endless support and love throughout my time at William and Mary.

LIST OF TABLES

2.1 Comparison: 5 World's Most Powerful Supercomputers [15]. 10

2.2 Comparison: Titan vs. Blue Joule [15]. 11

3.1 Specifications and Features of the Titan Supercomputer [17]. 22

3.2 Statistics for Temperature (°C) (DPR) 41

3.3 Statistics for Temperature (°C) (DBE) 45

4.1 Temperature mean retention time for cabinets in different temperature zones for GPU and CPU. 52

4.2 Statistics of temperature and power on Non-SBE offenders. 60

5.1 Precision and recall for three neural networks. 75

5.2 SBE Occurrence Prediction: Oracle vs. PRACTISE. 79

6.1 Precision and recall for basic schemes. 89

6.2 F1 score for SBE occurrence prediction. 93

6.3 Mean training time for various models. 94

6.4 Effect from temporal and spatial aspects of temperature and power features. 96

6.5 SBE occurrence prediction for "short-running" and "long-running" applications. 99

6.6 Percentage of correctly classified SBE-affected application runs in four severity levels. 99

7.1 Various metrics (including the total number of possible fault sites) related to considered GPGPU application kernels. 103
7.2 Fault sites and other statistics for GEMM. 110
7.3 CTA and threads groups for 2DCONV. 116
7.4 CTA and threads groups for HotSpot. 117
7.5 Effect of instruction-wise pruning for two threads. 120
7.6 Summary of instruction-wise pruning for selected kernels. Other kernels do not exhibit instruction commonality. 121
7.7 Statistics related to loops. 122

8.1 List of Applications with Evaluation Metrics and Thresholds. 146
8.2 The Impact of Different Inputs on CTA Group Popularity for HotSpot and RAY. **Notation: GRP-S/M/L=the percentage of CTAs in that group with Small/Medium/Large input, R=regular, IR=irregular.** 152
8.3 CTA-level and Warp-level Classification for Benchmark Kernels. **Notation: *%R-Grp.*= % regular groups over all groups, # *DI Grp.*=# of groups classified by dynamic instruction counts, # *ErrDist Grp.*=# of groups refined by fault distribution.** . 155
8.4 Resilience Coverage vs. Overhead Reduction. 164

LIST OF FIGURES

2.1 A representative CUDA-based GPU architecture. 12
2.2 A simplified overview of GPGPU application hierarchy. 13

3.1 Overview of the Titan supercomputer's physical organization. 22
3.2 Architecture of NVIDA K20X GPU deployed on the Titan supercomputer [9]. 23
3.3 Cumulative single bit error (SBE) count distribution over days (a), and cumulative SBE count distribution over days excluding top 2 days (b). 26
3.4 Daily SBE count across time excluding the top two days (a), and autocorrelation function of the SBE interarrival times (b). 27
3.5 Number of SBE-affected days for all nodes (sorted in increasing order of total SBE count) (a), and normalized variation in the daily SBE count distribution for the top twenty SBE offender nodes excluding the top two nodes (red line in the middle represents median while green dot represents mean) (b). 29
3.6 GPU resource distribution for the SBE offender nodes (excluding top two SBE offenders): GPU core hours (a), and GPU memory utilization (b). 30

3.7 Variance in the GPU resource utilization and daily SBE count: GPU core hours for top 50 days (a), for top 50 days excluding the top 3 days (b), GPU memory utilization for top 50 days (c), and for top 50 days excluding the top 3 days (d). Days are sorted in increasing order of SBE count. 31
3.8 GPU core-hours for users (a), and applications (b) experiencing SBEs. 32
3.9 DPR errors for SBE offender cards (excluding top two SBE offenders which had no DPRs). 34
3.10 Histograms of difference in SBE count for a 24-hour windows after and before the DPR occurrence for DPR offender nodes (a), and non-DPR offender nodes (b). Dotted vertical lines represent the average difference in SBE count. 35
3.11 Cumulative distributions of difference in SBE count for a 24-hour window (a) and a 72-hour window (b) for DPR offender nodes and non-DPR offender nodes. Some outliers are omitted for clarity. Omission of outliers causes the DPR-curve not to approach 1. 36
3.12 DPR affected GPU nodes with increasing error counts and normalized GPU core-hours (a), and normalized GPU memory utilization (b). . 37
3.13 DPR errors and GPU core-hours for DPR affected users (a), and DPR affected applications (b). 38
3.14 Distribution of execution time on random nodes, top 10 SBE nodes, and DPR offending nodes. 39
3.15 Temperature variation before each DPR occurrence. 43
3.16 Temperature variation before each DPR error. 44
3.17 CDF of temperature variation before DPR errors. 44

4.1 (a) Histogram of temperature for GPUs, CPUs, and DIMMs. The average temperature of GPU, CPU and DIMM are 28.1°C, 28.3°C and 26.1°C, respectively. The standard deviation of GPU, CPU and DIMM are 6.1, 9.4 and 2.6, respectively. (b) Monthly Histogram of GPU temperature. 50
4.2 *hot state*: retention time histogram for (a) GPU and (b) CPU. (Note that the long tail is truncated at 140min in both figures.) 51
4.3 *normal state*: retention time histogram for (a) GPU and (b) CPU. (Note that the long tail is truncated at 140min in both figures.) . . . 51
4.4 Weekly ranking for five hottest (a) and five coldest (b) cabinets. . . . 53
4.5 Non-uniform distribution of GPU error offender nodes at the cabinet level. 54
4.6 Non-uniform distribution of SBE-affected application runs at the cabinet level. 54
4.7 Workload and GPU error distribution: a small set of workloads experience most of the soft errors (a), and fraction of executions affected by SBEs for SBE-affected application runs (b). 55
4.8 Scatter plot of SBE count of SBE-affected application runs and their GPU utilization: core-hours (a) and memory (b). 56
4.9 Distribution of temperature (a) and power consumption (b) accumulative over the whole period at the cabinet level. 56
4.10 Temperature distribution of SBE offender nodes during SBE-free periods (a) and SBE-affected periods (b). Vertical lines represent mean values. 57
4.11 Power consumption distribution of SBE offender nodes during SBE-free periods (a) and SBE-affected periods (b). Vertical lines represent mean values. 58

4.12 Effect of neighboring components on temperature/power of an application over two runs on the same node overtime. Vertical solid lines represent the start and end of the aprun execution. 61

5.1 SBE occurrence prediction at the cabinet level. 76
5.2 Comparison between CDFs of ground truth, all prediction, and true positives for SBE occurrences at the cabinet level. 76
5.3 Autocorrelation and PRACTISE prediction for temperature. 78
5.4 Prediction for SBE occurrence at node level with PRACTISE. 79

6.1 *TwoStage* method: prediction flow. 89
6.2 Comparison of SBE occurrence prediction across different models for DS1. 92
6.3 Effect of different feature groups on F1 score, in terms of the improvement over *Basic A*. **All** means using all features discussed in Section 6.1. **Hist**, **TP**, and **App** correspond to SBE history, temperature/power consumption, and application-related features, respectively. 95
6.4 Decrement on F1 score if removing a certain feature set from the original feature combination: global vs local (a), and different length of SBE history (b). 97
6.5 Comparison between SBE occurrence prediction and ground truth at the cabinet level. 98

7.1 Overview of the 4-stage Fault Site Pruning Mechanism. 111

7.2 CTA grouping after $60K$ fault injection runs of one target instruction for (a) 2DCONV and (b) HotSpot. CTAs with the same color are classified into the same group. In the box plot, the horizontal green lines represent the median and red dots represent the mean. 114

7.3 CTA grouping given by average dynamic thread instruction count (*iCnt*) per CTA for (a) 2DCONV and (b) HotSpot. CTAs with the same color are classified into the same group. A significant similarity is is observed with Figure 7.2. 115

7.4 Thread Grouping inside one CTA. 118

7.5 PTXplus code comparison of two representative threads for PathFinder. Blue bold lines indicate common instructions. 120

7.6 Impact of loop-wise pruning on distribution of fault injection outcomes for (a) PathFinder, (b) SYRK, and (c)-(d) for K-Means K1 with different random seeds. 124

7.7 Distribution of fault injection outcomes of different bit position sections of two major register types (*.u32* and *.pred*) for (a) 2DCONV and (b) MVT. 125

7.8 Impact of bit-wise pruning on distribution of fault injection outcomes for (a) 2DCONV and (b) MVT (all registers). Percentage of outputs stabilizes at 16 bits. 126

7.9 Error resilience comparison of progressive fault site pruning techniques against the ground truth (baseline). 127

7.10 Fault site reduction comparison based on various feature-based pruning techniques. “+" indicates that each pruning stage is progressively built upon the pruned sites resulted from the previous stage. The height of the pruned fault sites bar is normalized by the original exhaustive fault sites for each application kernel, see last column of Table I. The effectiveness of progressive fault site pruning is compared against comprehensive baseline injection (60K random experiments). The exact numbers are shown on the top of the last two columns for the proposed method and the baseline case, respectively. 128

7.11 Error resilience comparison of progressive fault site pruning techniques against the ground truth (baseline) for (a) injecting double-bit faults and (b) injecting triple-bit faults. 134

7.12 Impact of increasing number of injected faults on the difference in terms of the percentage of (a) *masked*, (b) *SDC*, and (c) *other* outputs given by the proposed pruning technique and *baseline* for selected benchmark kernels. 135

7.13 Mean values of differences in terms of percentage numbers of different fault injection outcomes calculated (a) across all kernels and (b) across all kernels but excluding Gaussian K125. 136

7.14 Error resilience changes over increasing number of injected faults for representative benchmark kernels. 137

8.1 Effect of a single bit fault on the BlackScholes application output shows that a significant percentage of the fault injection runs lead to silent data corruption (SDC), which can be acceptable to a user (SDC-Accept). The percentage of SDC-Accept increases as the user-defined acceptability threshold becomes less conservative. 141

8.2 A high-level view of fault injection and evaluation framework. 147
8.3 Distribution of thread dynamic instruction (DI) counts at the CTA level for regular benchmarks (a) BlackScholes and (b) SCP. **The red triangle indicates the average and the blue error bar indicates one std.** 151
8.4 Distribution of thread dynamic instruction (DI) counts at the CTA level for irregular benchmarks (a) HotSpot and (b) RAY. **The red triangle indicates the average and the blue error bar indicates one std.** 151
8.5 Distribution of thread dynamic instruction counts at the warp level for SCP. **The red triangle indicates the average and the blue error bar indicates one std.** 153
8.6 Distribution of thread dynamic instruction counts at the warp level for MD k1, using two inputs: (a) Small and (b) Large. **The red triangle indicates the average and the blue error bar indicates one std.** 154
8.7 Distribution of fault injection outcomes at benchmark kernel level. (SDC faults are evaluated with the default threshold values.) 157
8.8 Impact of Small and Large inputs on fault distribution. 158
8.9 Error resilience characteristics at CTA level. **Each bar is distinguished by its group name and whether it is regular (R) or irregular (IR).** 160
8.10 Error resilience characteristics at warp level. **Each bar is distinguished by its group name and whether it is regular(R) or irregular(IR).** 161

8.11 Changes in the percentage of faults with increasing sample size for (a) BlackScholes and (b) SCP. *PCT.MSK*, *PCT.SDC*, and *PCT. OTR* indicate the percentage of masked, SDC, and other (including DDC, crashed, and hangs) faults, respectively. **Error bars give the 95% confidence intervals.** 162
8.12 Resilience coverage (% of *Masked* + *SDC-accept* outputs) as a function of different output quality. Output quality changes with different SDC acceptability thresholds. 166

9.1 The dynamic instruction count of threads in their launching order for HotSpot.. 170
9.2 The composition of threads with different dynamic instruction count inside each CTA for HotSpot. Each CTA contains 256 threads. None of the CTAs contains only one thread type while the colored part represents the dominant thread type. 170

Chapter 1

Introduction

Over the past decade, GPUs have become an integral part of mainstream high performance computing facilities. Parallelism provided by the GPU architecture has enabled domain scientists to simulate physical phenomena more quickly and accurately (i.e., at a finer granularity) [146, 79, 93] than what was previously possible by CPU-based large-scale clusters. Scientists are already benefiting from GPU deployment in large-scale computing systems such as the Titan supercomputer, the Blue Waters supercomputer, and the Keeneland cluster [146, 79, 69, 93]. Recognizing the performance and energy-efficiency benefits of GPUs, next generation pre-exascale supercomputers are also expected to continue taking advantage of parallelism provided by GPUs [13, 12]. Given the challenges of power provisioning for exascale systems, GPUs will continue to be an attractive choice due to their performance-per-watt characteristics that are better compared to their CPU counterparts [90].

Given the technology-trends and wide-spread adoption of GPUs, many researchers have studied the performance and energy-efficiency aspects of GPU-based applications in detail. In the meantime, the reliability of GPUs cannot be overlooked because most scientific applications are long-running, taking several hours or even days to complete. If software or hardware errors occur during application execution, they can not only lead to lower scientific productivity and operational efficiency of large-scale systems but can also

cause significant monetary loss [133]. Therefore, in addition to the principle of striving to achieve exascale performance at a stringent power budget, it is equal imperative to strive for more reliable GPU computing.

An initial step of any reliability study is to develop a deep understanding of GPU errors in the field. Computer architects have been investigating reliability characteristics of GPUs and ways to increase their reliability. Such efforts are often guided by technology projections and simplistic scientific kernels, and are performed using architectural simulators and modeling tools. Lack of large-scale field data impedes the effectiveness of such efforts. Only recently, researchers have started to investigate the reliability characteristics of GPUs using large-scale field data [140, 139, 92]. These recent studies have quantified the hardware and manufacturing-related failures of GPUs, firmware/application-related GPU errors, and how error resilience improves over generations of upgrades in GPU architecture. Still, as one of the most commonly observed errors, GPU soft errors (e.g., single-bit errors, double-bit errors, and dynamic page retirement errors), have not yet been studied well. There are many interesting aspects regarding GPU soft errors that are imperative to be explored, such as 1) the relationship between different types of GPU soft-errors, 2) the impact of soft-errors on application performance, and 3) the relationship between GPU soft-errors and user jobs, GPU resource utilization, temperature, and power consumption. Developing a better understanding on these aspects is the first focus of this dissertation.

Traditionally, the reliability of applications running on general-purpose GPUs is ensured with frequent check-pointing of application states [78] and error correction and detection codes for major GPU storage components (i.e., register files, shared memory, caches, and main memory). However, these protection mechanisms often come at high performance, power, and area costs [74, 155, 156, 75, 76]. For example, the impact of error-correcting code (ECC) overhead on real-world computational science applications can be as high as 10% on GPUs [26]. Nevertheless, the decreased memory bandwidth caused by ECC overhead can result in larger performance degradation than the decreased fraction of bandwidth itself due to queuing. Facing such expensive reliability overhead in

large-scale systems, computational scientists already naively turn off ECC for their application runs [53]. Still, completely turning off error protection can be too risky. In such cases, a prediction model that can accurately forecast the occurrences of GPU soft-errors would be useful in guiding flexible error protection mechanisms for GPU nodes, e.g., by dynamically turning on/off error protection based on prediction. Acknowledging the necessity of an error predictor, the second focus of this dissertation is to elaborate on the challenges, process, and solutions involved in building effective error prediction models. Specifically, we consider features that are related to the occurrences of GPU soft errors guided by our large-scale reliability analysis in the field. We observe that the relationship is rather complex and is non-trivial to be exploited by straight-forward statistical models. Therefore, we leverage machine learning models to capture such complicated interactions between system/application features and the prediction target (i.e., soft-error occurrences). Through our evaluation, the proposed models are able to accurately forecast the occurrences of soft errors on GPU nodes.

The aforementioned characterization and prediction efforts towards GPU soft errors are performed at the system level. Such coarse-grained reliability analysis is beneficial in terms of acquiring an overview on the GPU soft errors in the field and of developing system-level solutions to improve system reliability. Still, such analysis has its inherent drawbacks. First, large-scale system measurement data in the wild are *post hoc* and we have limited control over the data collection methodology, i.e., our analysis is subject to the data and information provided by Titan's system administrators. Second, we have no knowledge regarding the applications running on the system, except for their binary executable file names. Unfortunately, the lack of application-related information impedes us from any further exploration towards specific application types. To understand the effects of soft errors on specific applications, we do fine-grained reliability analysis on a wide set of popular GPGPU benchmarks to understand why applications react to GPU soft errors differently, that is we aim to explore the different *error resilience characteristics* of GPGPU applications. This is the third focus in this dissertation.

There is a rich body of prior work studying the error resilience characteristics of GPU applications [154, 85, 44, 59]. These works propose different kinds of fault-injection models to systematically inject faults to diverse locations (i.e., registers) in applications (commonly referred to as *fault sites*) and evaluate the tolerance of applications in the presence of GPU errors. One of the major challenges in building an effective fault-injection model is fault sites selection, i.e., injecting faults in all possible fault sites and explore their effect. Suppose we consider a single-bit flip model that injects one fault per application execution, which is the de-facto model used in studies in this area [154, 44, 59] and is shown to be effective and sufficient in capturing the error resilience characteristics of GPU applications [124], the total number of exhaustive fault injection sites for benchmarks in commonly used benchmark suites (i.e., CUDA [110], Rodinia [31], and Polybench [54]) can range from millions to even billions, such as 3.44×10^7 for HotSpot [31] and 6.23×10^8 for GEMM [54], and 6.38×10^9 for BlackScholes [110]. The tremendous size of fault sites is due to the fact that each GPGPU application kernel can spawn hundreds to thousands of threads and each thread is assigned to a dedicated amount of on-chip resources (e.g., register files, ALUs, and shared memory). For the calculation of fault sites reported above, we only consider soft errors that occur in functional units (e.g., arithmetic logic unit and load-store unit), which are not protected in current commercial GPUs [4, 8, 10]. Yet, the number of fault sites is tremendous. Executing one experiment per fault site in such a vast space to collect application reliability metrics is clearly impossible and absolutely not practical.

Consequently, it is imperative to first resolve this challenge of how to systematically and efficiently reduce the number of fault sites required in a fault-injection campaign. Facing this challenge, prior works [85, 44, 59] mainly resort to statistical solutions, such as randomly selecting a number of fault sites based on the desired error margin and confidence interval [83]. Such statistical methods have two limitations. First, a large number of fault sites is required to deliver statistically significant results. For example, it is necessary to conduct $60K$ experiments to achieve a confidence interval of 99.8% and an error mar-

gin of 1.26% [83]. Second, this method gives no intuition about fault site selection from the architecture perspective, such as whether different GPGPU application resources react similarly or differently to faults. In this dissertation, we take an orthogonal approach, by pruning the large amount of fault sites via carefully considering the properties of GPGPU applications. By carefully selecting pruning mechanisms, we aim to reduce the total number of required fault sites while still maintain accuracy in capturing GPGPU application resilience characteristics. Naturally, this framework can serve as a tool to deepen the understanding on the error resilience characteristics of GPGPU applications.

Note that, this framework is build on the commonly used single-bit fault model [154, 44, 59], the next natural step is to extend it to multi-bit fault model, especially for GPGPU applications. Previous studies have looked into the impact of multi-bit faults for CPU applications [124]. However, GPGPU applications normally spawn many more threads than CPU applications, making it non-trivial to directly apply those techniques to GPGPU applications. Therefore, in this dissertation, we also investigate the impact of multi-bit faults on the outcomes of GPGPU applications by extending the proposed framework to the multi-bit fault model.

Lastly, inspired by the proposed framework, we devise a hierarchical approach to study the error resilience characteristics of GPGPU applications at three different levels, including kernel levels, CTA levels, and warps levels. We observe that CTAs (or warps) show different resilience features. In other words, some CTAs (or warps) are very error resilient while others are sensitive to soft errors. In addition, we notice that not all corrupted application outputs due to soft errors are unacceptable to the end users. If users are willing to sacrifice some output quality, there would be sufficient opportunities for providing low-overhead and reliable GPGPU error protection strategies.

1.1 Contributions

The contributions of this dissertation are summarized as follows:

- *System-level GPU reliability analysis* [101, 102, 103]: We conduct a large-scale study on GPU soft-errors on a real-world in-production HPC system – the Titan supercomputer, which is America's fastest supercomputer for open science [18]. We discover features with indicative capability of GPU errors and exploit them for error occurrence prediction.

- *Application-level GPU reliability analysis* [104, 100]: We investigate the effect of GPU errors on application execution by first proposing a systematic way of progressively pruning the fault site space for a doable and practical fault injection campaign.

1.1.1 GPU Reliability Analysis at the System Level

- We perform a large-scale study on GPU soft-errors on the Titan supercomputer, including single-bit errors (SBEs), double-bit errors (DBEs), and dynamic page retirement errors (DPRs). We investigate their characteristics and relationship with GPU resource utilization, applications, users, and temperature, as well as the relationship between different types of errors [101].

- We conduct a deep exploration on the most commonly observed errors – single-bit errors (SBEs) [102, 103]. We discover that workload characteristics, certain GPU cards, temperature and power consumption have complex interaction with GPU SBEs.

- We propose two effective machine-learning-based predictors that are able to accurately forecast the occurrences of GPU SBEs [102, 103]. More specifically,

  - We show how to systematically select input features for prediction models by categorizing them into spatial and temporal dimensions [103].

  - We acknowledge the presence of imbalanced dataset in the Titan traces and overcome this challenge with a customized similarity-reduction-based algorithm that is capable of under-sampling the majority class [102].

- We devise a neural-network-based prediction framework [102] to forecast GPU SBE occurrences.
- We devise a two-stage prediction framework by taking advantage of the inherent dataset features and evaluate its effectiveness across several machine learning models [103].

1.1.2 GPU Reliability Analysis at the Application Level

- We quantify the problem of high number of fault sites in the fault-injection campaign for GPGPU applications [104].
- We develop progressive pruning techniques by leveraging GPGPU-specific properties, such as similarity in terms of resilience characteristics in threads, commonality in common code blocks, presence of large portion of loop iterations, and relationship between resilience features and location of bits in the registers. With the proposed solution, we are able to reduce the fault space by up to seven orders of magnitude while maintaining accuracy that is close to the ground truth [104].
- We extend the proposed fault site pruning technique to multi-bit fault model and evaluate the accuracy via various GPGPU benchmarks.
- We come up with a hierarchical approach to study the error resilience characteristics of GPGPU applications at various levels, including kernel level, CTA level, and warp level. We observe that different CTAs (or warps) exhibit different error resilience characteristics [100].
- Giving the fact that not all corrupted GPGPU application outputs are unacceptable to the user, we are able to strike the balance between reducing error protection overhead and preserving acceptable application output quality at the kernel, CTA, and warp levels [100].

1.2 Organization

This dissertation is organized as follows. In Chapter 2, we present the background and basic concepts that are used in this dissertation. In Chapter 3, we present a large-scale study of soft-errors on GPUs in the field [101]. In Chapter 4, we focus on the most commonly observed GPU soft-errors, that is single-bit errors (SBEs), and investigate their complex interaction with several related features [102, 103]. In Chapter 5 and 6, we introduce two machine-learning-based solutions to predict SBE occurrences [102, 103]. In addition to the study of GPU reliability at the system level, in Chapter 7 we turn to understand the resilience features of GPGPU applications, by starting with designing an effective fault injection framework with progressively pruned fault sites, for both single-bit and multi-bit fault models [104]. Then, in Chapter 8, we show a hierarchical approach to understanding error resilience characteristics of GPGPU applications at kernel, CTAs, and warps levels and illustrate opportunities of enabling low-overhead yet reliable GPGPU computing through a use case [100]. Finally, in Chapter 9, we describe future research directions.

Chapter 2

Background and Related Work

In this chapter, we introduce basic concepts and terminology that are used in the entire dissertation. First, we present the prevalent adoption and the benefits of accelerators, especially GPUS, on large-scale systems (see Section 2.1). Then, in Section 2.2, we explain the background knowledge related to GPUs. Lastly, we discuss related work on GPU reliability analysis in Section 2.3.

2.1 Benefits of Accelerators on Large-Scale Systems

Recently, supercomputers have been providing powerful computational capability for scientific applications from diverse domains, such as science, engineering, medicine, social media, gaming, and finance [41, 119, 136, 46, 125, 112, 106, 109]. For years, researchers keep pushing the envelop on the maximum and peak computational speed of supercomputers, reflecting as the constantly fluctuating ranks on the Top500 list [18]. On the other hand, supercomputers are normally costly. They require huge power for computation and on cooling. Consequently, power efficiency (i.e., performance-per-watt) is becoming important complement metric to compare supercomputers, yielding in another fierce competition in Green500 list [15].

Table 2.1 presents the 5 most powerful supercomputers in the world (as of November, 2017) [18, 15]. Surprisingly, we notice that Tianhe-2 (launched in 2013) and Titan

(launched in 2012) still provide superior computational capability that outperform a lot of newly-launched systems. From the table, we make several observations:

Table 2.1: Comparison: 5 World's Most Powerful Supercomputers [15].

Top500/ Green500 Rank	Name	Country, Year	Total Cores	Rmax [TFlop/s]	Rpeak [TFlop/s]	Power (kW)	Power Effeciency [GFlops /Watts]	Accelerator/ Co-Processor
1/20	Sunway Tai-huLight [48]	China, 2016	1.1e+07	9.3e+04	1.3e+05	1.5e+04	6.05	None
2/137	Tianhe-2 [16]	China, 2013	3.1e+06	3.4e+04	5.5e+04	1.8e+04	1.90	Intel Xeon Phi 31S1P
3/10	Piz Daint [11]	Switzer-land, 2017	3.6e+05	2e+04	2.5e+04	2.3e+03	10.40	NVIDIA Tesla P100
4/5	Gyoukou [6]	Japan, 2017	2.0e+07	1.9e+04	2.8e+04	1.4e+03	14.17	PEZY-SC2 700Mhz
5/105	Titan [17]	USA, 2012	5.6e+05	1.8e+04	2.7e+04	8.2e+03	2.14	NVIDIA Tesla K20x

Rmax - Maximal LINPACK performance achieved; Rpeak - Theoretical peak performance.
Rank is as in November, 2017.

First of all, except the top first supercomputer – Sunway TaihuLight, all others are assisted with accelerators or co-processors to boost achievable performance. Note that, the Top500 list is ranked by *Rmax* (i.e., maximal performance achieved). When looking at *Ppeak* (i.e., theocratical peak performance), Sunway TaihuLight is beaten by all the other four, including the two launched in 2012/2013. This indicates the powerful performance boosting ability provided by accelerators/co-processors.

Most importantly, if we focus on the power efficiency (i.e., performance per watt), we further acknowledge the importance of deploying accelerators/co-processors. Gyoukou and Piz Daint are very power-efficient by achieving a score over 14.17 GFlops/Watts and 10.40 GFlops/Watts, respectively. They mainly benefit from accelerators. Piz Daint is equipped with NVIDIA Tesla P100 [14], a powerful general-purpose GPU that is specially designed for compute-intensive and high-parallel applications. Gyoukou uses PEZY-SC2 [34], a very close cousins of GPU chips. Moreover, it is worth mentioning that Titan (with power efficiency of 2.14 GFlops/Watts) and Tianhe-2 (with power efficiency of 1.90 GFlops/Watts) surprisingly beat Sunway TaihuLight (with power efficiency of 6.05 GFlops/Watts). In

fact, if we look at supercomputers launched in 2012, Titan has the close-to-best power efficiency but superior computational capability, as compared with the most power-efficient supercomputer Blue Joule at that time (see Table 2.2). A noticeable difference between the two supercomputers is that Titan accelerates by GPUs while Blue Joule does not. These evidences show that accelerators/co-processors play an important role in building power-efficient supercomputers.

Table 2.2: Comparison: Titan vs. Blue Joule [15].

Top500/ Green500 Rank	Name	Country, Year	Total Cores	Rmax [TFlop/s]	Rpeak [TFlop/s]	Power (kW)	Power Effeciency [GFlops /Watts]	Accelerator/ Co-Processor
5/105	Titan [17]	USA, 2012	5.6e+05	1.8e+04	2.7e+04	8.2e+03	2.14	NVIDIA Tesla K20x
88/92	Blue Joule [2]	UK, 2012	1.3e+05	1.4e+03	1.7e+03	6.6e+02	2.18	None

Rmax - Maximal LINPACK performance achieved; Rpeak - Theoretical peak performance.
Rank is as in November, 2017.

Inspired by the success of deploying accelerator/co-processors on high-performance computing systems, two more powerful next-generation supercomputers are being deployed – Summit [13] on Oak Ridge National Lab and Sierra [12] on Lawrence Livermore National Lab, both are expected to in operation in 2018.

Acknowledging the necessity of accelerators/co-processors in achieving both powerful computational ability and power efficiency, we focus in this dissertation on the most commonly-used accelerators– general-purpose GPUs (or GPUs in short) [18, 15].

2.2 General-Purpose GPUs for Scientific Computing

In this section, we introduce the concepts and background knowledge for general-purpose GPUs (or GPUs for short). Throughout the dissertation, we use CUDA-based terminology (CUDA stands for Compute Unified Device Architecture [1]) created by NVIDIA as NVIDIA GPUs are widely-used in the field. In fact, among the supercomputers with

accelerators/co-processors in Top500 List, over 85% are boosted with NVIDIA Tesla GPUs [18].

2.2.1 Baseline GPU Architecture

Figure 2.1 shows a representative CUDA-based GPU architecture. A typical GPU consists of multiple simple cores, also called streaming-multiprocessors (SMs) in NVIDIA terminology [111]. Each core is associated with private L1 data, texture and constant caches, software-managed scratchpad memory, and register files. Cores are connected to memory channels (partitions) via an interconnection network. Each memory partition is associated with a shared L2 cache, and its associated memory requests are handled by a GDDR5 memory controller. Recent commercial GPUs, i.e., Fermi [8], Kepler [10] and Pascal [4], use unified single-error-correction double-error-detection (SEC-DED) error correction codes (ECCs) to protect register files, L1/L2 caches, shared memory and DRAM from soft errors, and use parity to protect the read-only data cache. Other structures like arithmetic logic units (ALUs), thread schedulers, instruction dispatch unit, and interconnect network are not protected.

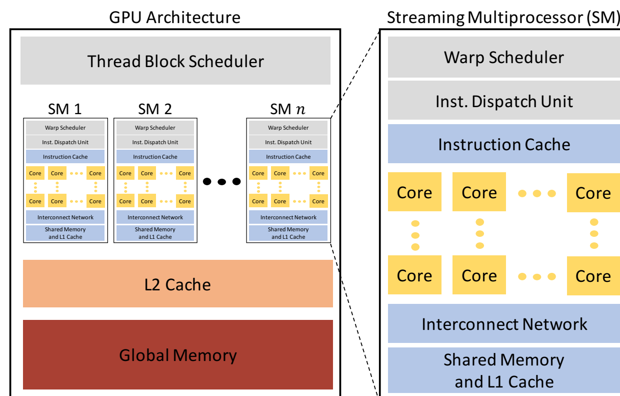


Figure 2.1: A representative CUDA-based GPU architecture.

2.2.2 GPGPU Applications and Execution Model

GPGPU applications rely on the single-instruction-multiple-thread (SIMT) philosophy and concurrently execute thousands of threads over large amounts of data to achieve high throughput. Figure 2.2 presents a simplified overview of thread hierarchy in GPGPU applications. A typical GPGPU application execution starts with the launch of kernels on the GPU. Each kernel is divided into groups of threads, called *thread blocks*, which are also known as *Cooperative Thread Arrays* (CTAs) in CUDA terminology. A CTA encapsulates all synchronization and barrier primitives among a group of threads [77, 67]. Having such an abstraction allows the underlying hardware to relax the execution order of the CTAs to maximize parallelism. The underlying architecture sub-divides each CTA into groups of 32 individual threads (called warps) that execute a single instruction on the functional units in lock step. This sub-division is an architectural abstraction and is transparent to the application programmer.

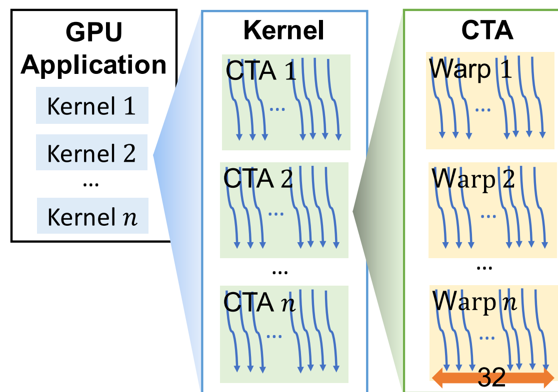


Figure 2.2: A simplified overview of GPGPU application hierarchy.

2.2.3 GPU Errors in the Field

GPU errors can be classified into several categories. GPU hardware related errors, such as double bit errors, off-the-bus errors, and micro-controller halts cause the application to crash. Soft errors that can be corrected by the ECC mechanism do not result in execution

loss. Single bit errors are corrected by the SECDED ECC. Two single bit errors on the same page result in a dynamic page retirement (DPR) error [108]. This particular error is also reported when a double bit error happens and the page is retired in order to improve the longevity of the card.

There is a host of GPU related errors including errors that are caused by the application, driver issues, firmware bugs, or thermal issues. Note that NVIDIA documents a list of such XID errors and their possible causes [19]. GPU applications may also terminate with a non-zero exit code, indicating that the execution was not successful. Other than hardware-related and XID errors, several other reasons may be responsible for non-zero exit codes, e.g., programming errors and expiration of time-quota. Prior works [55, 139] study these XID errors and system-integration errors (e.g., Off the Bus).

2.3 Reliability Analysis in the Field

In the Section 2.1, we give an overview of the performance/power efficiency benefits of deploying GPU accelerators in supercomputers. Reliability of such large-scale systems is equally worthy of research attention, especially since the scientific applications running on these systems are typically long-running [140, 139]. Any software or hardware error that occurs during application execution decreases the scientific productivity and operational efficiency and could result in significant monetary loss [133]. Therefore, it is important that applications are able to cope with different types of runtime failures and errors. As we progressing towards exascale, applications are going to face even more severe resilience challenges, due to the increasing number and decreasing size of the components required for exascale systems [90]. In this section, we discuss past work on reliability analysis from two perspectives: large-scale system level (see Section 2.3.1) and application level (see Section 2.3.2).

2.3.1 System-Level Reliability Analysis

Quantifying and characterizing system failures is key to improving the reliability of any computing system. This is even more important for large-scale computing systems since the impact of system failures is large on these system and may lead to significant scientific productivity and monetary loss. Consequently, researchers have investigated failures on large-scale systems in detail [92, 42, 88, 89, 113, 117, 123, 128]. Several studies have exploited the insights from such efforts to predict failures and adapt fault-tolerance mechanisms to minimize the impact of system failures. Some of these studies propose to predict failure by identifying the correlation among failure events [49, 50, 51, 88]. Such proposals often rely on machine learning and other prediction techniques on the RAS logs and the system logs. This may result in high-overhead and low lead time for prediction, but nevertheless they demonstrate that failure prediction is possible and effective in certain cases.

Several studies have focused on studying the reliability aspect of large-scale computing systems. For example, Liang et al. investigated different component failures including network, disk, memory and CPU for the Blue Gene/L system, and proposed failure prediction models [88]. Oliner et al. investigated system failure logs for multiple HPC systems at the Los Alamos National Laboratory and the Sandia National Laboratory, including Red-Storm and Thunderbird system [113]. They studied both software and hardware errors and developed a methodology for applying filtering of failure logs. Schroeder et al. have studied the system failures and its impact on multiple HPC systems at LANL [128].

There have also been more failure studies for a given system component such as DRAM, disks, and SSDs. For example, DRAM-focused efforts have shown the effect of vendors on soft-errors [62, 129, 135]. These studies also show the pitfalls in studying the DRAM errors and its impact on the reliability assessment of the system. Disk-focused studies demonstrate that disk failures in the field can be significantly higher than what one would estimate from the vendor's sheet [126, 23]. Such studies also show that peripheral components fail

more often than one may expect in large scale storage systems. Recent studies [95] on SSD failure in the field provide insights about differences in the early detection life cycle between SSDs and Disks, lack of read disturbance error in the wild, and implication of these findings for future SSDs. However, large-scale GPU reliability characterization studies have been relatively limited [57, 92, 140], primarily because the GPU architecture is a relatively newer technology.

Recently, there have been efforts focusing on studying and improving GPU reliability at scale [25]. Several recent studies [92, 140, 139] present error characterization for the GPU-enabled Cray supercomputers such as the NCSA Blue Water and Titan supercomputer. They study the spatial and temporal characteristics of GPU errors, how these errors propagate spatially in a short time-window, frequency of GPU errors in different memory structures of a GPU, correlation between batch jobs and correctable GPU errors, etc. These efforts have primarily focused on understanding XID errors, manufacturing errors (e.g., Off the Bus error), and its effect on application-execution. These studies have also shown via neutron beam testing that more recent generation of GPUs are more error resilient than previous generation of GPU architecture. These studies have also focused on issues and challenges with current GPU error logging methods. Previous efforts by Haque et al. [57] have deployed a software-based GPU soft-error detector on Folding@home distributed platform for two different architectures, the G80 and GT200 architectures. They showed that newer generation of GPUs observed significantly lower soft error rate. Additionally, they found that the GPUs were sensitivity to memory faults in a pattern-dependent manner.

Limitations of prior work: None of the aforementioned studies present detailed analysis and characterization of soft-errors on GPUs at large-scale, especially the most commonly observed single-bit errors. It is important to understand the complex interplay between GPU errors and related factors (i.e., workloads, resource utilization, temperature, and power consumption). Such understanding helps strike a balance in reducing resilience overhead and preserving reliability, i.e., by building effective error prediction frameworks.

2.3.2 Application-Level Reliability Analysis

Besides the preceding system-level reliability studies, researchers have also leveraged simulation-based analysis to detect critical hardware structures that are more vulnerable to soft errors. In particular, prior works [45, 64, 138] have conducted architectural vulnerability analysis (AVF), which tracks every bit in an architecture during the application run and calculates the likelihood of the bit that can affect the output. In addition, fault-injection models are also effective in understanding the impact of faults. Although there is a large body of prior work on fault injection models/frameworks [92, 42, 88, 89, 113, 117, 123, 128, 126, 23, 95, 49, 50, 51] in the context of CPUs, only a limited set of fault injector models have been proposed for GPUs. Yim et al. [154] built a source-to-source translator, SWIFI, to investigate error resilience in GPUs and demonstrate that the ratio of silent data corruption (SDC) in GPUs is much higher than that observed in CPUs. In order to capture the impact of faults at the architecture level, Fang et al. [44] developed GPU-Qin that leverages the GPU debugging tool *cuda-gdb* [3] to inject one fault into the destination operand or the address operand of arithmetic and memory instructions. However, since GPU-Qin uses *cuda-gdb*, it cannot inject faults into control flow instructions. Hari et al. [59] addressed this problem with a compiler-based fault injection framework, SASSIFI, which injected single bit errors into the destination operand of any kind of assembly instructions.

Limitations of prior work: Prior work mainly focuses on developing fault injection models using compiler-based or simulation-based tools. In terms of fault sites selection, these works mostly randomly sample a tiny subset of all potential fault injection locations in the application, whose size is often in the magnitude of millions to billions, to capture a view of the overall resilience characteristics for GPGPU applications [83]. For example, with 95% confidence interval and error margin of 6%, it is necessary to launch around 1000 fault injection runs. The number increases to $60K$ when seeking a more strict requirement, i.e., 99.8% confidence interval and error margin of only 1.26% [83]. However, such fault

site selection methodology impedes any deeper resilience analysis at the kernel, CTA, and warp levels. In addition, it would be also interesting to see if we can deliver the same resilience characteristics with fault sites with less than $60K$ or even less than $1K$ samples, which is the focus of this dissertation.

2.4 Chapter Summary

In this chapter, we first show the benefits given by the prevalent adoption of GPUs in large-scale systems. Then, we discuss prior efforts in understanding the reliability of GPUs that motivate and guide the research presented in this dissertation. Finally, we explain useful GPU-related knowledge.

Chapter 3

A Large-Scale Study of Soft-Errors on GPUs in the Field

In this chapter, we describe a large-scale investigation on GPU soft-errors in the filed by analyzing large amount of measured system related data on the Titan supercomputer, which is America's fastest supercomputer for open science [17]. We are especially interested in characterizing GPU soft-errors in the field, focusing on their impact on application performance, their relation to user jobs, GPU resource utilization, and temperature, as well as the relationship between difference kinds of GPU soft-errors. The goal of this chapter is to improve our understanding in the aforementioned aspects and identify observations that are instructive to future system design and resource management.

Unfortunately, there are several challenges in building such an understanding. First, there are often multiple factors responsible for different types of GPU errors, making it hard to distill their cause and their impact on applications. Second, it is hard to study the correlation or impact of applications on GPU reliability characteristics or resource utilization since we do not have access to the end-user application-base. Third, we often do not have control over several factors such the power/cooling conditions, user behavior or node-assignments to different jobs. This makes the development of an accurate understanding of the GPU errors more challenging. Despite these challenges, in this chapter,

we attempt to improve our current understanding about GPU reliability at-scale while carefully considering these challenges.

Specifically, we quantify and characterize the soft-errors on the Titan supercomputer's GPU nodes. This chapter uncovers several interesting and previously unknown insights about the characteristics and impact of soft-errors (e.g., single bit error, dynamic page retirement error, and double bit error). We characterize the temporal characteristics of single bit errors and its association with other errors. We study the impact of workloads, resource utilization, and variance in load-level on error-affected GPU nodes. In particular, our study aims to understand the correlation between application characteristics and specific GPU errors. Our study also provides a deep understanding of possible temperature effects on soft-errors. As we describe our findings, we also point out how different methodologies may lead to different observations, and the importance of our observations to system administrators and architects. We believe that insights obtained from our large-scale field data analysis carry significant implications for current and future HPC computing facilities, system operators, and system architects.

This chapter is organized as follows. In Section 3.1, we discuss on related work. Section 3.2 presents the organization of the Titan supercomputer, the data collection methodology, and the limitation and scope of this study. In Sections 3.3 and 3.4, we investigate the characteristics of single-bit errors (SBEs) and dynamic page retirement errors (DPRs) on the Titan supercomputer, respectively. We explore the application performance variance on soft-error-affected GPU nodes in Section 3.5 and the relationship between temperature and GPU soft-errors in Section 3.6. Finally, we offer brief chapter conclusions in Section 3.7.

3.1 Related Work

There is a rich body of work in the literature in understanding the reliability of large-scale distributed systems [92, 42, 88, 89, 113, 117, 123, 128, 126, 23, 95, 49, 50, 51].

The focus of these studies is to identify failures of different components such as network, disk, memory and CPU, analyze their impact on HPC systems, and predict future failure events. Sridharan et al. [134] conducted a large-scale study on DRAM failures on the Jaguar supercomputer at Los Alamos National Lab and found that soft faults account for around 30% of all kinds of DRAM failures. Siddiqua et al. [131] suggested that this percentage is smaller based on the data collected from the studies performed on a variety of large-scale data centers. Recent large-scale studies in the wild [92, 140, 139] reveal that modern GPU architectures suffer from reliability shortcomings. Martino et al. [92] and Tiwari et al. [140, 139] characterized the statistical characteristics of GPU failures and errors on the Blue Waters supercomputer at the University of Illinois and the Titan supercomputer at Oak Ridge National Laboratory, respectively, and show that GPU error rates are non-negligible.

The above works target all kinds of GPU errors. Given the fact that GPU soft-errors are the most commonly observed errors in the Titan supercomputer, it is imperative to specifically focus on soft-errors on GPU nodes. In this chapter, we discover several previously unknown insights about the characteristics of GPU soft-errors, such as single-bit errors, dynamic page retirement errors, and double-bit errors. For the first time, we characterize the temporal characteristics of single bit errors and their association with other errors. In contrast to previous works, we investigate the impact of workloads, resource utilization, and variance in load-level on error-affected GPU nodes in detail. This chapter also provides a deep understanding of the temperature effects on soft-errors in GPUs. Given that GPUs are likely to be an important part of an exaflop HPC system, we believe that our study with the America's largest GPU-enabled system would help the community in improving the understanding the impact of GPU errors on scientific applications and their implications for large-scale GPU resource management.

3.2 Methodology

In this section, we provide an overview of Titan and its GPU architecture. Next, we provide details about our data collection and analysis methodology. We also describe the limitations of this study.

3.2.1 Titan Supercomputer Organization and NVIDIA K20X GPU Architecture

Fig. 3.1 shows the physical organization of the Titan supercomputer. It consists of 200 cabinets organized in 25 rows and 8 columns. Each cabinet has three cages/chassis. There are eight blades in each cage. Four nodes constitute one blade. Each node has one AMD Opteron 6274 CPU (with 32 GB of DDR3 memory) and one NVIDIA K20X GPU (with 6 GB of GDDR5 memory). Each blade has two high-speed interconnect Gemini routers, each shared by two nodes. Table 3.1 lists Titan system specifications and features [17].

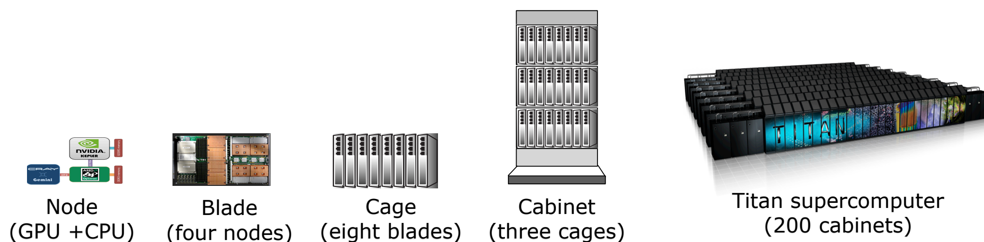


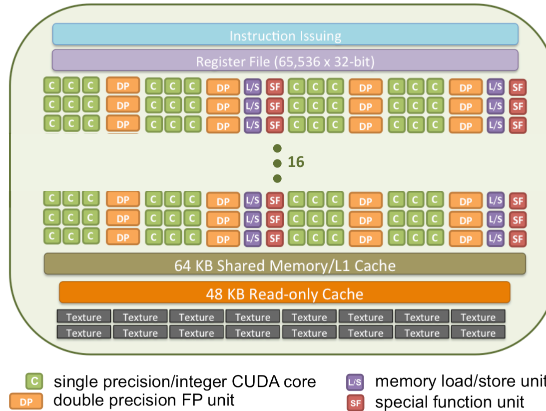
Figure 3.1: Overview of the Titan supercomputer's physical organization.

Table 3.1: Specifications and Features of the Titan Supercomputer [17].

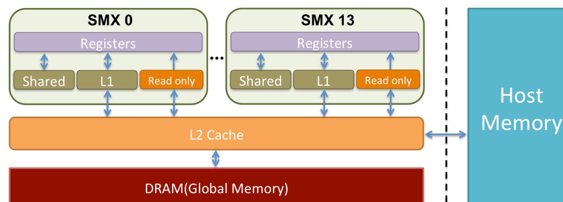
Architecture	Cray XK7	**Memory/node**	32GB + 6GB
Processor	16-Core AMD	**Memory/core**	2GB
Cabinets	200	**Interconnect**	Gemini
Nodes	18,688 AMD Opteron	**GPUs**	18,688 K20X Keplers
Cores/node	16	**Speed**	27 PetaFlops
Total cores	299,008 Opteron Cores		

The GPU deployed on the Titan supercomputer is NVIDIA K20X GPU [9], which is

able to provide a peak performance of over 1.30 Tflops (double precision). The K20X GPU has a total of 14 streaming multiprocessors. Figure 3.2(a) presents the architecture of one streaming multiprocessor. Each streaming multiprocessor has 192 CUDA cores, 64K registers, 64KB of combined shared memory and L1 cache, and 48KB of read-only data cache. Figure 3.2(b) shows the memory hierarchy of K20X GPU. We see that the L2 cache (1536 KB) and the GDDR5 memory (6GB) is shared by all streaming multiprocessors. On-chip and off-chip GPU memory structures including the device memory, L2 cache, instruction cache, register files, shared memory, and L1 cache are protected by a Single Error Correction Double Error Detection (SECDED) error correction code (ECC). The read-only data cache is parity protected. On the other hand, structures such as logic, thread schedulers, instruction dispatch unit, and interconnect network are not protected by the ECC.



(a) Architecture in the Streaming Multiprocessor



(b) K20X GPU Memory Hierarchy

Figure 3.2: Architecture of NVIDA K20X GPU deployed on the Titan supercomputer [9].

3.2.2 GPU Errors: Collection and Analysis Methodology

In this chapter, we study GPU soft-errors that occurred on 18,688 GPUs deployed on the Titan supercomputer, including single bit errors (SBEs), dynamic page retirement errors (DPRs), and double bit errors (DBEs). We use GPU-error related data from February 2015 to June 2015 (more than 60 million node hours). The console logs from Titan are parsed to log critical system events. These critical system events alert the system operators of unexpected/undesired behavior. We point out that we apply a filter to separate a "parent" failure event from its "child" events. This methodology is similar to the one outlined in previous works [55, 140, 141, 113], but understanding the impact and effect of "parent/child" failure events is not the focus of this chapter, this topic is covered in detail by other works [55, 139].

We note that single bit errors are not logged in the console log, these errors are collected via the *nvidia-smi* utility on all GPU nodes. This utility provides snapshot information, i.e., it does not timestamp individual single bit errors, but records single bit errors before and after each batch job. This allows us to do temporal analysis on single bit errors, albeit at the granularity of a "batch job". We denote a batch job as a set of applications that are submitted by the same user (using a qsub command on Titan). Multiple "applications" (also referred as "apruns") can run within a submitted batch job (also referred to as "job" or "batch"). The single bit error count is collected at the start and end of the batch job and hence, can not be associated with an application run directly. We also note that our framework can identify the node locations on which the single bit errors occur. We collect GPU resource utilization information such as GPU core-hours, maximum memory consumption, and total memory consumption, on a per application basis.

The output from the nvidia-smi utility also includes double bit and dynamic page retirement related errors. We do not use this utility to analyze double bit or dynamic page retirement errors due to inconsistency in error logging as pointed out by previous works [140].

3.2.3 Limitations and Scope

While our study covers the GPU error data for a supercomputing facility over an extended period of time, we recognize that our work is subject to assumptions and limitations.

First, our analysis is post-hoc in nature and hence, by definition, it can not answer what-if scenarios where one may require changing the system/workload environment to observe the effect of a change.

Second, we note that such a large-scale computing facility is often very dynamic in nature with respect to software stack changes. Operational practices are continuously tweaked and unscheduled outages take place among other system updates. We have limited control over such factors. Therefore, isolating the impact of the above factors on our study is challenging. Instead, as we discuss our findings in this chapter, we specifically point out the external factors that we believe may influence our findings. Previous works have also pointed out that NVIDIA's GPU error logging has improved over time [140, 139]. Our error collection framework attempts to mitigate this by collecting the same error information via multiple possible methods.

Third, our study provides insights about correlation between applications/users and GPU error characteristics. Yet, it is not possible to investigate specific applications since we do not have access to application source codes. We point out that we have little to no knowledge about users' intentions. User behavior may change over time as the scientific knowledge in a particular domain improves. A new computational model or method in a particular domain may affect all applications in that domain at a given time or over a period of time. We also note that while our logs report the application name (binary name) at the end of each job, it is possible for a user to use the same binary name for two different applications, or the same application with different input types. For our analysis, we conservatively treat them as the same application because of the lack of more detailed knowledge.

3.3 Analyzing Single Bit Errors (SBEs)

In this section, we aim to understand the temporal characteristics of single bit errors (SBE) on the Titan. While previous studies have shown that most of the SBEs tend to occur only in a few GPU cards [140], the temporal characteristics of the SBEs have not been explored because of the inability to collect SBE occurrence information continuously over time. As described earlier in Section 3.2, our framework enables us to collect SBE counts at the batch job granularity.

Fig. 3.3(a) shows the CDF of the single bit error counts on a per day granularity. Recall that the time stamp of each SBE occurrence is not recorded. However, since the Titan supercomputer is highly utilized, we are able to collect the SBE data from a large number of batch jobs and aggregate them over 24-hour periods. The x-axis in Fig. 3.3(a) presents the days in the observation data in increasing order of their daily SBE count.

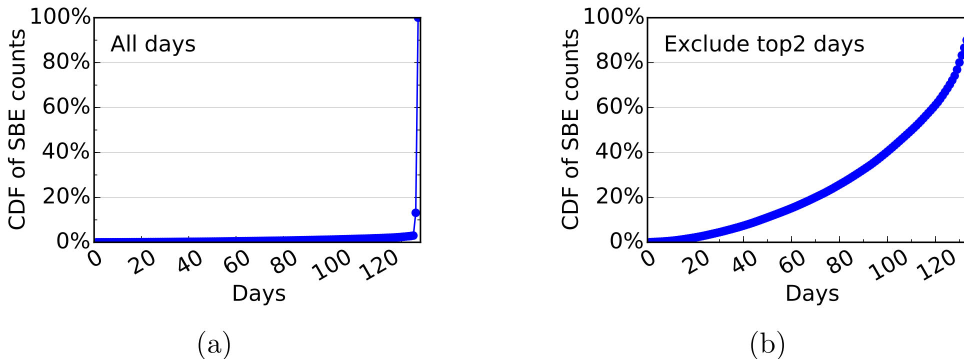


Figure 3.3: Cumulative single bit error (SBE) count distribution over days (a), and cumulative SBE count distribution over days excluding top 2 days (b).

The steep curve of the distribution suggests that only a few days account for most of SBEs. In fact, only three days account for 97.18% of the total SBEs, while the top ten days with most SBEs account for 97.84% of the total SBEs. Due to this skewness, it not clear how errors are being accumulated over the rest of the days. To better view this, we plot in Fig. 3.3(b) the cumulative distribution function of SBE counts but exclude the top two days. We observe that SBE occurrences are not proportionally distributed over the

rest of the days either, i.e., 40% of the days with the lowest SBE daily counts account for only 10% of the total SBE counts, while the remaining 60% of the days account for 90% of SBEs.

This uneven distribution of SBEs across days led us to investigate how these errors appear across time. Fig. 3.4(a) shows the normalized SBE count *per day* for the whole period of the study. We normalize the daily SBE count by the average of the daily SBE count over the whole period. This figure indicates that the density of SBEs across days is fairly uneven and appears bursty.

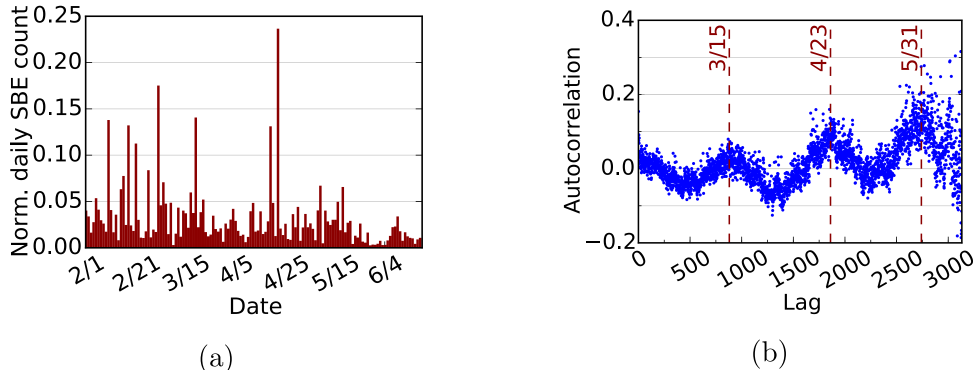


Figure 3.4: Daily SBE count across time excluding the top two days (a), and autocorrelation function of the SBE interarrival times (b).

To examine whether there is burstiness and/or periodicity in SBEs, we analyze the time series of SBE occurrences and plot the autocorrelation function of the inter-arrival times of SBE-affected batches with non-zero SBE counts since SBE measurements are at the per batch granularity. Autocorrelation is a mathematical representation of the degree of similarity in a time series and a lagged version of itself. As such, it is ideal for discovering repeating patterns by quantifying the relationship between different points of a time series as a function of the time lag [82]. The autocorrelation metric is in the range of $[-1, 1]$. Higher positive values indicate that the two points between the computed lag distance are "similar", i.e., have stronger correlation. Zero values suggest no periodicity. Negative values show that the two points that are lag elements apart are diametrically different. Fig. 3.4(b) illustrates the autocorrelation function of the inter-arrivals of batches as a function of the

distance between successive arrivals (lags). The figure illustrates a noticeable periodic pattern: the pattern repeats within every 6 weeks, and the periodic pattern for positive autocorrelation values becomes even more pronounced as the lag increases. This indicates that both burstiness and periodicity are present, it may be therefore possible to predict future SBE occurrences using this information[151]. One may argue that burstiness in SBE occurrence is an artifact of burstiness in the inter-arrival of GPU jobs. To address this, we performed the autocorrelation analysis on the number of applications executed every day. We found the autocorrelation metric to be close to zero, indicating lack of burstiness in the inter-arrival of GPU applications. This is expected since the Titan supercomputer is a highly-utilized computing platform with long job-queue waits. Therefore, we conclude that our observation about SBEs in not an artifact of the GPU job execution characteristics.

Observation 3.1 *Our field data analysis suggests that single bit error occurrences on the Titan supercomputer are bursty in nature. These errors tend to be clustered in time. Given that most of these errors are also limited to only a few GPU cards [140], system administrators can exploit these observations together for better GPU job scheduling at a large scale (e.g., avoid scheduling critical workloads on certain nodes / days, and possibly turn off ECC on certain nodes during specific time-periods for improved performance).*

Previous work has shown that only a few selected GPU cards experience most of the SBEs in the system [140]. We note that the measurement period here does not overlap with that of a previous study on the same system but, our study reconfirms the findings presented in previous work [140]. Here, the top two SBE offenders out of all 590 SBE offenders account for 96.9% of SBE errors. Interestingly, we also found that these top two SBE offenders accumulate all the SBEs on a single day. This leads us to investigate how SBE offender nodes accumulate these errors over time and look for how many distinct days each SBE offender experiences one or more SBEs. Fig. 3.5(a) shows the number of distinct days that a specific SBE offender node experiences an error. We make two observations.

First, as illustrated by the points in the bottom right corner of the plot, a few top SBE offender nodes experience most of their errors over a small number of days. Second, the rest of nodes do not show a linear trend in terms of the number of distinct days over which SBEs occur. For example, the bottom 65% of the SBE offenders (approximately 400 nodes) accumulate their SBEs over less than 20 days, while the top 35% of SBE offenders (approximately 200 nodes) take up to 6 times more days to accumulate their SBEs. This non-linearity in SBE accumulation can be particularly useful to HPC facility administrators for identifying high SBE offender nodes and exploiting this information for better GPU job scheduling.

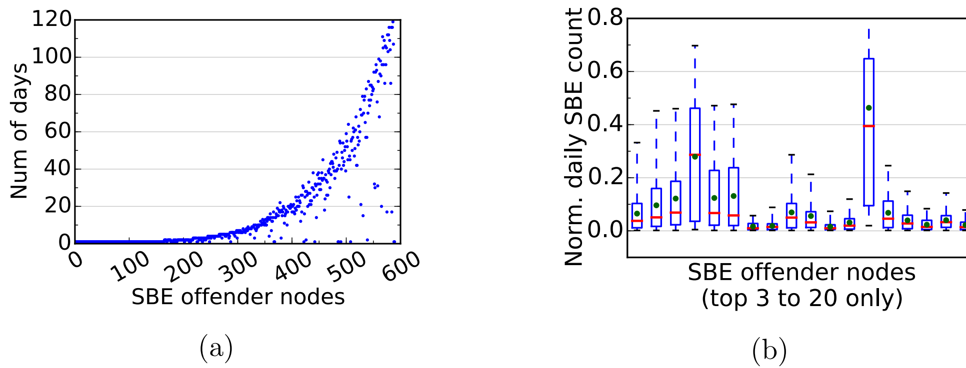


Figure 3.5: Number of SBE-affected days for all nodes (sorted in increasing order of total SBE count) (a), and normalized variation in the daily SBE count distribution for the top twenty SBE offender nodes excluding the top two nodes (red line in the middle represents median while green dot represents mean) (b).

Motivated by the above observation, we look deeper into the top 20 SBE offender nodes. In particular, we plot the variation in daily SBE count for the top 3 to 20 nodes (we do not consider the top 2 nodes because all their SBEs occur on a single day only). Fig. 3.5(b) illustrates the boxplot of the daily SBE counts that shows the 25th and 75th percentiles as well as median (flat line) and mean (dot). The boxplots show that variation can be significantly high for certain nodes. This suggests that while high count SBE offenders accumulate single bit errors over a large number of distinct days, it may be challenging to predict the number of single bit errors these nodes are expected to experience on a particular day.

Observation 3.2 *A few top SBE offenders experience all of SBEs over a very small number of days. However, the rest of nodes do not show a linear trend in terms of the number of distinct days over which SBEs occur. High count SBE offenders experience errors over a significantly high number of distinct days compared to the low count SBE offenders nodes. Moreover, the variation of SBE occurrence among days can change significantly across SBE offender nodes.*

After investigating the temporal characteristics of SBE occurrences, we attempt to understand how GPU resource utilization affects SBE occurrences. In particular, we test if higher GPU resource utilization may lead to higher SBEs. We point out that single bit errors can occur due to multiple reasons, therefore, higher GPU resource utilization alone may not be considered as the "cause". Fig. 3.6 shows the normalized GPU core hours and memory utilization for all SBE offender nodes. The normalization is performed using the average for all SBE offender nodes except the top two nodes (which are considered outliers, as their SBEs occur in a single day only). We observe that the nodes with higher SBE count do not necessarily use higher GPU core hours or run workloads with higher memory utilization.

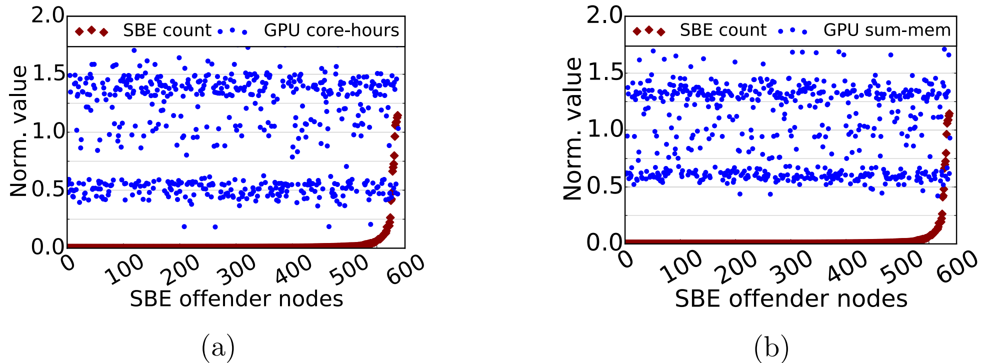
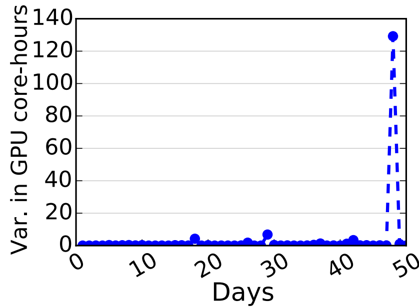
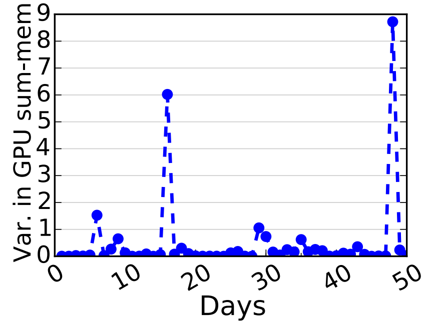


Figure 3.6: GPU resource distribution for the SBE offender nodes (excluding top two SBE offenders): GPU core hours (a), and GPU memory utilization (b).

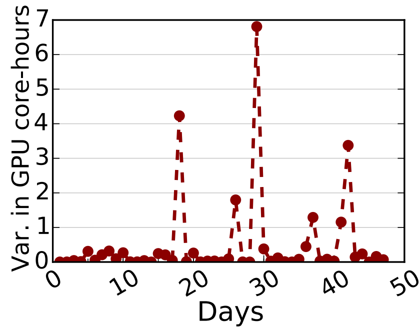
While GPU resource utilization does not seem to be directly correlated with the SBE occurrence frequency on the GPU nodes, we suspect that the variance in GPU resource



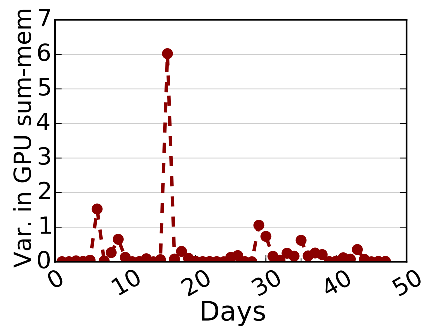
(a) top 50 days



(b) top 50 days



(c) top 4 to 50 days



(d) top 4 to 50 days

Figure 3.7: Variance in the GPU resource utilization and daily SBE count: GPU core hours for top 50 days (a), for top 50 days excluding the top 3 days (b), GPU memory utilization for top 50 days (c), and for top 50 days excluding the top 3 days (d). Days are sorted in increasing order of SBE count.

utilization may be correlated to higher SBE occurrences. More precisely, we want to test the hypothesis that days with higher variance in GPU utilization experience higher single bit errors. Fig. 3.7 shows the top 50 days that encountered most SBEs (in increasing order) and the corresponding variance in GPU resource utilization on that day. We note that Figs. 3.7(a)-(b) indicate that the couple of days with the highest SBE count may also experience the highest variance in their GPU resource utilization. However, a closer look at the top 4 to 50 days (Figs. 3.7(c)-(d)) shows that variance in GPU resource utilization does not imply higher daily SBEs.

Observation 3.3 *We found that GPU resource utilization and the variance in the GPU resource utilization do not seem to be significantly correlated with the SBE oc-*

currences. Higher GPU resource utilization or its variance do not necessarily result in a higher SBE count. We believe that an important implication of this finding is that GPU resilience simulation and modeling frameworks do not necessarily need to vary the soft-error rate based on the compute load or variance in the load. This can potentially simplify the design of such tools without compromising the accuracy of the study.

We learned that the GPU resource utilization is not highly correlated with the SBE frequency on SBE offender nodes. Here, we investigate the relationship between specific users/applications and SBE counts. In other words, is a certain fraction of users/applications experiencing more single bit errors than others? If so, what are the respective GPU resource utilization levels?

Fig. 3.8(a) shows the SBE count of different users versus their respective GPU core hours. Both SBE count and GPU core hours have been normalized by their respective average values. We also point out that only users that encountered at least one single bit error are included in the plot. We found that the correlation between GPU core hours and SBE count is significant when studied at the user-level. The Pearson coefficient is 0.59 with p-value < 0.05 while the Spearman coefficient is 0.89 with p-value < 0.05. This indicates a strong non-linear correlation. We did similar analysis between the SBE count for users versus their respective GPU memory utilization. We found similar trends in the results.

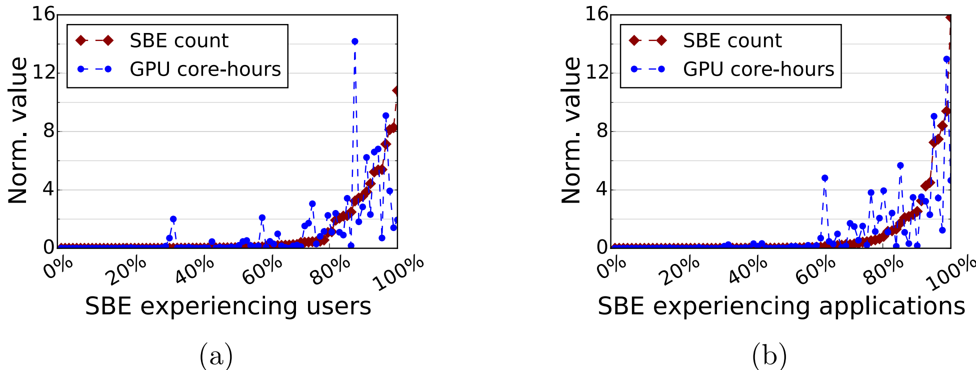


Figure 3.8: GPU core-hours for users (a), and applications (b) experiencing SBEs.

Fig. 3.8(b) shows that SBE count for applications versus its respective GPU core hours. Only the applications affected by SBEs are included in the plot. Similar to our previous analysis for users, we found strong non-linear correlation in this case as well. The Pearson coefficient is 0.67 with p-value < 0.05 while the Spearman coefficient is 0.89 with p-value < 0.05. Analysis between the SBE count of different applications versus their respective GPU memory utilization shows similar trends.

In summary, our data suggests that GPU resource utilization at the user-level appears highly correlated with the SBE frequency for different users and applications.

Observation 3.4 *SBE occurrence frequency appears to be highly correlated with users and applications. This correlation is better expressed by a non-linear relationship and is not necessarily an artifact of the bursty nature of single bit errors. This indicates the necessity and importance of application-centric GPU error resilience techniques and tools.*

3.4 Analyzing Dynamic Page Retirement (DPR) Errors on the Titan Supercomputer

Dynamic Page Retirement (DPR) is an important resilience feature to improve the longevity of an otherwise good GPU card. A page in the GPU device memory is blacklisted if two single bit errors or one double bit error occur on the same page. This page is not allocated to the application on the next reload of the GPU driver [108]. In this section, we study single bit errors and dynamic page retirement errors together since SBEs can cause DPRs. We also investigate the impact of GPU resource utilization and applications on DPR occurrence.

For the measurement period, we observe a total of 50 DPR errors on 43 distinct GPU cards. Recall that we observe that SBEs tend to be more concentrated in a few selected GPU cards. More generally, the distribution of SBEs is not uniform among all the 590 SBE offender cards. Therefore, we hypothesize that DPR errors are more likely to occur in

the top SBE offender cards. Fig. 3.9 shows the DPR and SBE error frequencies for all SBE offender cards (excluding the top 2 SBE offenders which do not have any DPR). The plot shows that some top SBE offender cards do observe DPR errors. For example, the top 10 SBE offender cards account for 4 DPR errors, while the top 20 SBE offender cards account for 7 DPR errors out of total 50 DPRs. Cards with low SBE counts show no DPRs during the measurement period. Most of the top SBE offenders do not experience any DPRs either. It is possible that some top SBE offender cards may potentially experience a DPR error in the future but we argue that our measurement period is long enough to account for most of such cases given the bursty occurrences of SBEs.

Figure 3.9: DPR errors for SBE offender cards (excluding top two SBE offenders which had no DPRs).

Observation 3.5 *Top SBE offender GPU cards do not necessarily experience more dynamic page retirement errors. In fact, DPR errors may occur on any SBE offender cards, even to those with relatively lower single bit error counts.*

One can also reasonably hypothesize that the SBE count is likely to be higher on DPR offender nodes before the DPR error, since two SBEs trigger a DPR. To test this hypothesis, we calculate the difference of SBE counts after and before each DPR occurrence within a certain time window (i.e., 24-hours), for both DPR offender nodes and non-DPR offender nodes. In other words, we accumulate the SBE count on the node for a 24 hour window both after and before the DPR event, and then take their difference. Fig. 3.10 presents

the histograms of the difference in SBE count for a 24-hour window for both DPR offender nodes and non-DPR offender nodes. The dotted vertical line in each graph shows the average. Average value of this difference for DPR offenders is around 160 while the value for non-DPR offenders is around 0. Similarly, the cumulative distribution in Fig. 3.11(a) shows that DPR offending nodes and non-DPR nodes have significantly different distributions. We also conduct the Kolmogorov-Smirnov Test (KS test) to test this hypothesis. We find that D = 0.389, p-value = 5.991×10^{-7}. For our sample size here, the critical D value is 0.19 and therefore we can reject the null hypothesis, and conclude that DPR offending nodes show significantly higher values of difference in SBE counts compared to non-DPR nodes.

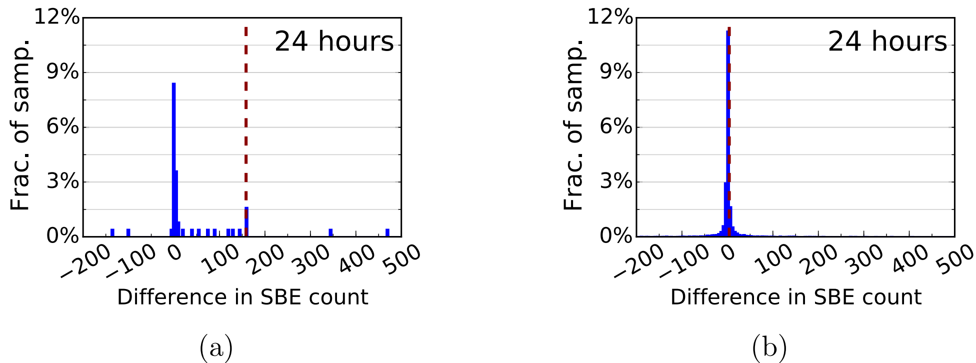


Figure 3.10: Histograms of difference in SBE count for a 24-hour windows after and before the DPR occurrence for DPR offender nodes (a), and non-DPR offender nodes (b). Dotted vertical lines represent the average difference in SBE count.

Next, we test if SBEs continue to occur on the DPR offender nodes beyond the 24-hour period since the last DPR error occurrence. If so, for how long do the DPR offender nodes continue to experience single bit errors? Fig. 3.11 shows the cumulative distributions of difference in SBE count for two different size of time windows. As a comparison point, we present results for 24 hours time-window and 72 hours time-window (Fig. 3.11(a) and (b)). We observe that the cumulative distribution does not change significantly from 24 hours to 72 hours. This indicates that the majority of SBEs occurring after the DPR occurrence tend to occur within the first 24 hours. We find that the likelihood of SBEs increases after

the DPR occurrence, but it does not continue to remain at that level always. We also note that the time-period after which the probability of SBE occurrence returns to normal level can vary across GPU nodes. We find 24 hours to be a good threshold. In summary, this is an interesting and counter-intuitive finding as the original hypothesis suggests higher SBE occurrences before the DPR error; on the contrary, our field data indicates that more SBEs are likely to occur after the DPR error.

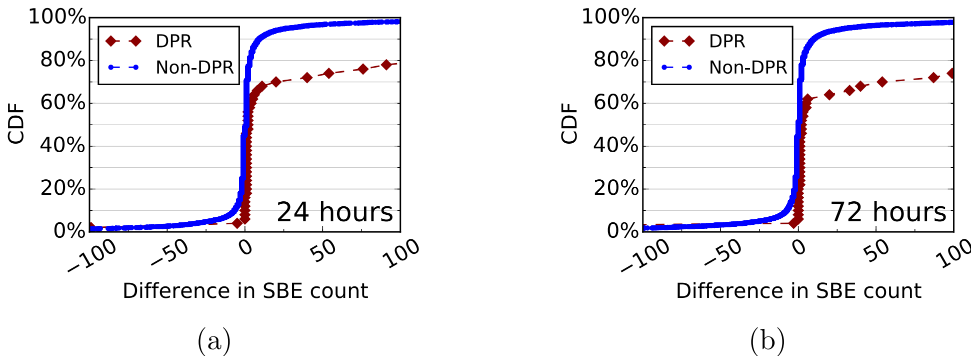


Figure 3.11: Cumulative distributions of difference in SBE count for a 24-hour window (a) and a 72-hour window (b) for DPR offender nodes and non-DPR offender nodes. Some outliers are omitted for clarity. Omission of outliers causes the DPR-curve not to approach 1.

Observation 3.6 *Our field data analysis shows that single bit errors tend to occur more frequently on the DPR offender nodes after the DPR error than before. This is counter-intuitive since single bit errors are a cause of DPR errors, and hence, one would expect the SBE error rate to be higher before the DPR error. We also observe that the majority of SBEs occurring after the DPR occurrence tend to occur within first 24 hours. This finding can be useful in cases where an application/user may turn on/off ECC support based on the probability of soft-error occurrences.*

Recall that the DBE is another cause for DPR errors. We conduct analysis to understand the relationship between DBEs and DPRs. However, due to the limited number of errors, it is not possible to draw conclusions with high statistical significance. Next, we investigate the effect of GPU resource utilization on the DPR error frequency, similar to

the analysis performed for single bit errors. Fig. 3.12 presents the GPU resource utilization for the GPU nodes that experience DPR errors. We point out that the GPU core-hours and sum-memory metrics are normalized to the corresponding average across *all* nodes. Fig. 3.12 shows that GPU resource utilization points do not show any clear trend. Nodes that experience a DPR do not have higher resource utilization compared to nodes that do not experience DPR errors. This finding is similar to the one expressed in Fig. 3.6 where SBE events do not show strong association with the GPU resource utilization.

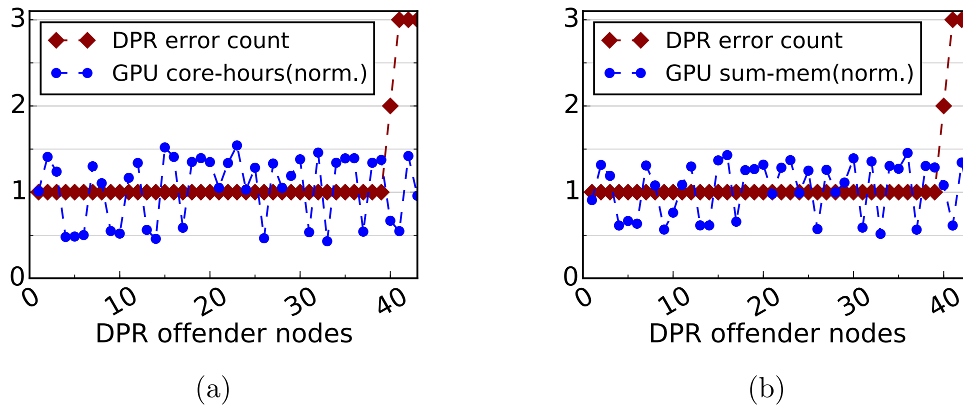


Figure 3.12: DPR affected GPU nodes with increasing error counts and normalized GPU core-hours (a), and normalized GPU memory utilization (b).

Observation 3.7 *There is no significant association between DPR count and GPU resource utilization.*

As we do not find any significant relation between GPU resource usage on DPR affected GPU nodes and DPR error frequency, we now look into how GPU resource usage of certain users and applications correlates to DPR errors. Naturally, the GPU resource usage varies among different users and applications. Therefore, we investigate if applications that experience higher DPR errors also have higher GPU resource utilization. Fig. 3.13 shows that GPU resource utilization is not necessarily correlated to the number of DPR events experienced by different users and applications. The Spearman and Pearson coefficients

show almost no correlation. In summary, we can observe from Fig. 3.13 that users that experience more DPR errors do not necessarily use longer GPU hours.

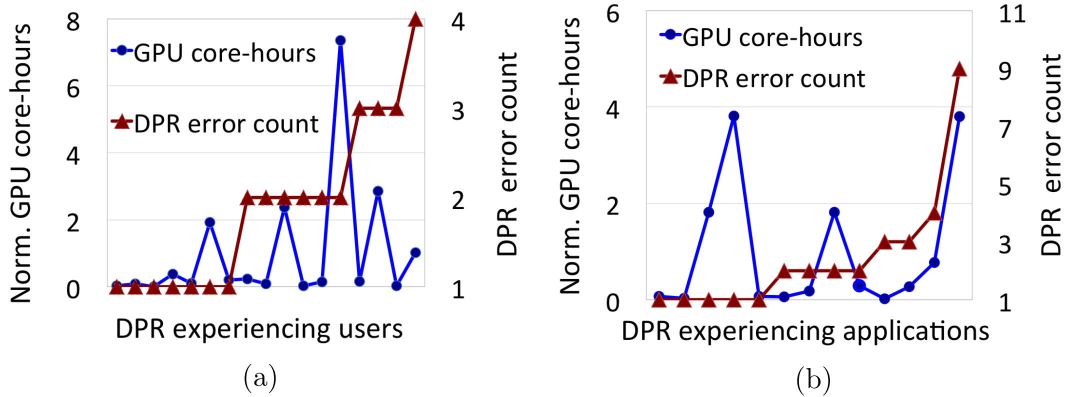
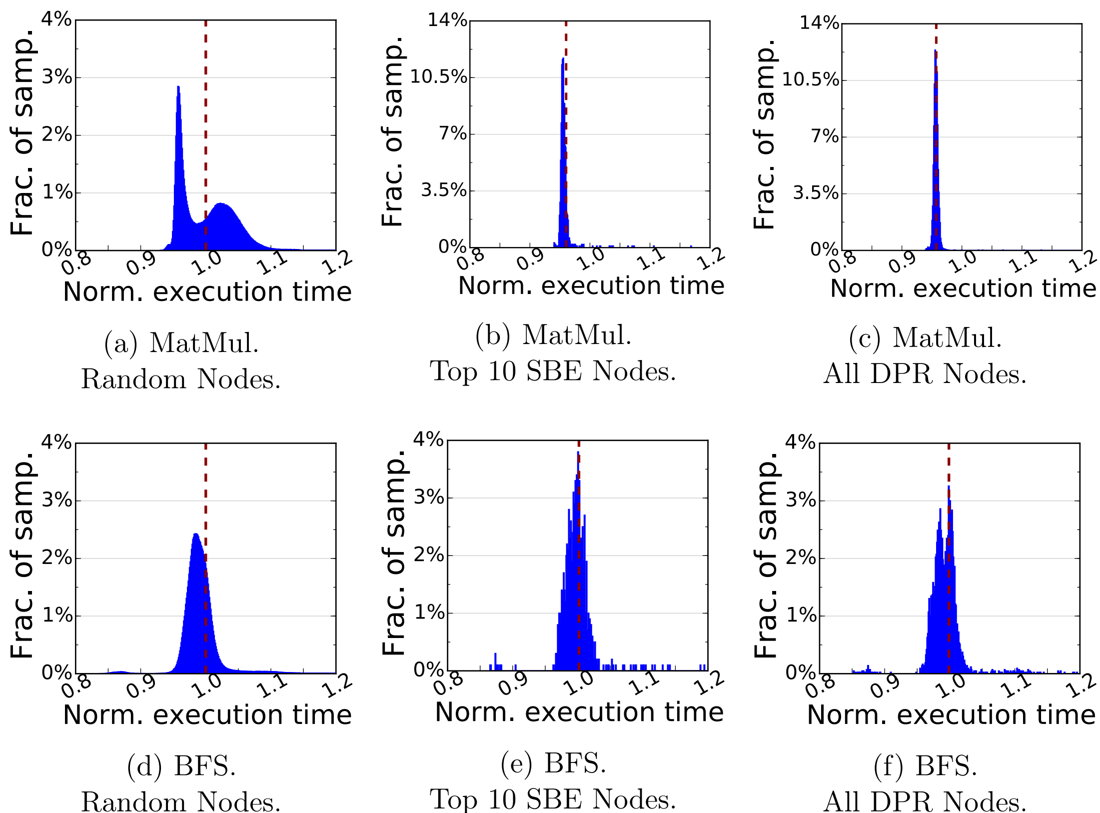


Figure 3.13: DPR errors and GPU core-hours for DPR affected users (a), and DPR affected applications (b).

3.5 Analyzing Performance Variance in SBE and DPR Affected GPU Nodes

In this section, we investigate if nodes affected by SBE and DPR errors are more likely to show higher performance variation or significant degradation in performance compared to error-free nodes. Toward this, we perform extensive experiments on the SBE and DPR affected nodes and randomly selected nodes on Titan.

We run two representative GPU kernels, Matrix Multiplication (MM) and Breadth-first Search (BFS) on all DPR nodes, top 10 SBE offender nodes, and randomly selected error-free GPU nodes. These kernels have significantly different computational characteristics. MM is a regular, compute-intensive benchmark, while BFS is an irregular data-intensive one. The MM and BFS kernels were obtained from the NVIDIA CUDA toolkit [107] and Rodinia Benchmark Suite [31], respectively. We collect performance data by repeatedly running these kernels on the selected GPU nodes. We conducted over 24,000 experiments on Titan GPU nodes, covering more than 9,000 randomly selected GPU nodes. Each kernel



(a) MatMul. Random Nodes.
(b) MatMul. Top 10 SBE Nodes.
(c) MatMul. All DPR Nodes.
(d) BFS. Random Nodes.
(e) BFS. Top 10 SBE Nodes.
(f) BFS. All DPR Nodes.

Figure 3.14: Distribution of execution time on random nodes, top 10 SBE nodes, and DPR offending nodes.

is run 100 times on each DPR offender node and top 10 SBE offender nodes.

Fig. 3.14 shows the distribution of execution times on randomly-selected nodes, top 10 SBE offender nodes, and DPR offender nodes. The execution time on the x-axis is normalized with respect to average performance across all runs. We note that some outliers in these plots are omitted for presentation clarity but their effect on mean and standard deviation is reflected on the graphs. For the MM benchmark, we notice that SBE nodes and DPR nodes have 3-4% better performance on average compared to the randomly selected nodes. This is because randomly-selected nodes exhibit a bimodal distribution of execution times, making the average execution time of these nodes slightly higher. For the BFS benchmark, there is no significant difference in average performance between the top 10 SBE and DPR offender nodes compared to randomly-selected nodes. The SBE

and DPR offending nodes show slightly lower standard deviation compared to randomly selected Titan nodes. This is primarily because the number of DPR and top 10 SBE offender nodes are much smaller compared to our randomly-selected node pool (over 9,000 nodes). There are other factors that can cause higher standard deviation among such a large number of nodes (e.g., variance in temperature, spatial location, device properties).

Observation 3.8 *The distribution of execution time across randomly selected nodes on Titan in itself may be application-dependent. Experimental data suggests that top 10 SBE and DPR offending nodes do not exhibit lower performance than the average performance of randomly selected nodes. The implication of this finding is that system operators do not need to replace GPU cards with high SBE / DPR error counts specifically for performance degradation or variance reasons.*

3.6 Effect of Temperature on Dynamic Page Retirement Errors

In this section, we investigate the effect of temperature on GPU soft-errors, in particular DPRs, DBEs, and SBEs. Past work points to temperature dependence of hardware errors on other systems [127, 43]. Here, we perform a detailed analysis of the relationship between temperature and soft-errors on GPUs.

In the Titan supercomputer, upper cages are typically at higher temperature than lower cages. We found that the distribution of DPR errors across different levels of cages is fairly equally distributed. Therefore, this does not imply a direct impact of temperature on DPR errors as such. To investigate deeper, we collected GPU card-level temperature for different time windows of 5 minutes, 15 minutes, and 60 minutes *before* each DPR occurrence for a large number of GPU nodes. We collect temperature data every minute for each GPU card. Table 3.2 shows the mean and standard deviation of temperatures across the three time windows of 5 minutes, 15 minutes, and 60 minutes before each DPR

occurrence. These statistics are collected for the DPR offender node, all nodes within its cage, and over 800 random nodes in the system.

Table 3.2: Statistics for Temperature (°C) (DPR)

	60min before (avg / stddev)	**15min before** (avg / stddev)	**5min before** (avg / stddev)
DPR	34.55 / 8.53	37.00 / 8.95	39.02 / 8.79
Non-DPR (same cage)	34.68 / 8.71	36.77 / 9.24	38.56 / 9.27
Non-DPR (random)	30.54 / 7.70	31.54 / 7.79	32.47 / 8.11

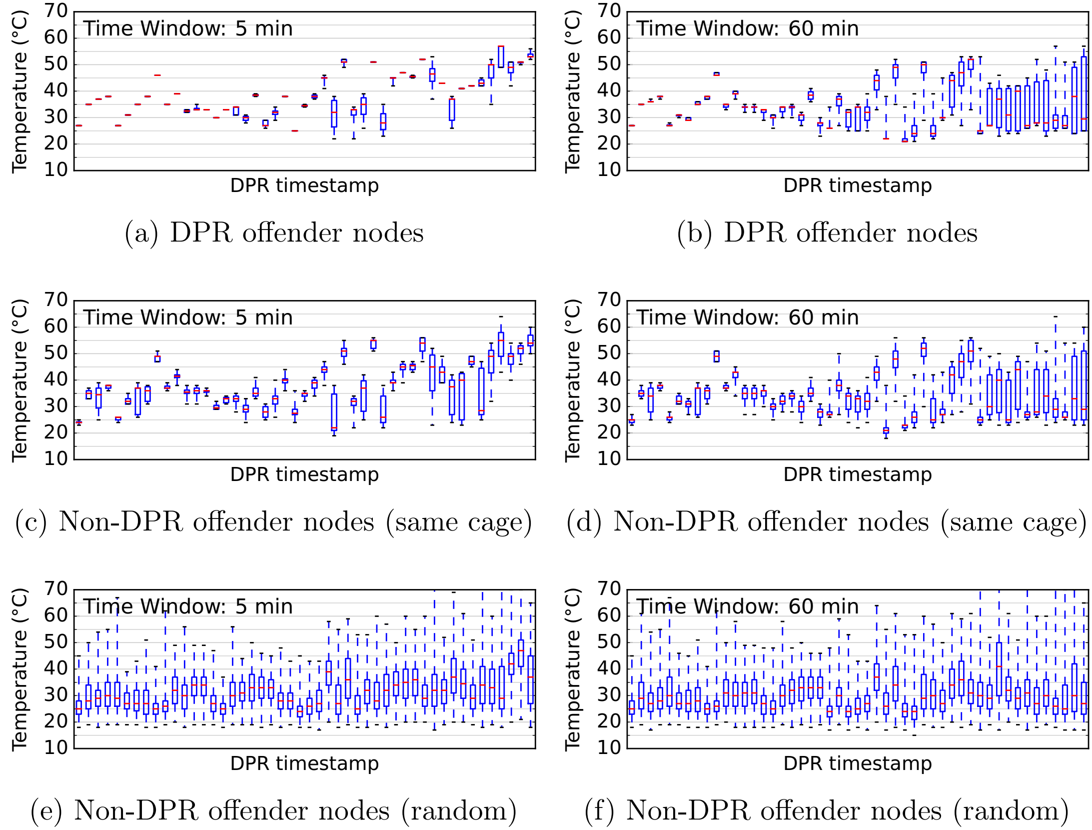
First, we observe that temperature across all three types of nodes increases consistently during the hour as the DPR occurrence approaches (Table 3.2). This may be possibly due to power/cooling condition in the machine room or the currently running workload. Interestingly, the DPR offenders have higher average temperature than randomly selected nodes. This indicates that higher temperature may be associated with DPR errors. We also note that the nodes in the same cage as the DPR offenders show similar average temperature. This suggests that higher temperature may be associated with the increase in the likelihood of a DPR error. However, one can not trivially conclude that higher temperature leads to DPR errors since other GPU nodes in the same cage do not observe a DPR error despite similar average temperature.

These results emphasize the importance of selecting the correct methodology for comparisons: comparing the data across random nodes in the entire system and nodes within the same cage, we see the importance of selecting what to compare with. Choice of random nodes may sound as the right choice for comparison but in such-large scale systems usage behavior and node characteristics can be significantly different in randomly chosen nodes.

Table 3.2 also shows that the standard deviation in temperature across all three types of nodes is similar. Yet, the standard deviation for DPR offenders is generally higher than the standard deviation for randomly selected nodes. Since the standard deviation is a

single number and may not capture the entire picture, we investigate deeper to understand the effect of temperature variation of DPR errors. Fig. 3.15(a)-(f) illustrate how temperature variations occur for different type of nodes with a DPR occurrence for the extreme time windows: 5 minutes and 60 minutes. Each element on the x-axis corresponds to a DPR occurrence (i.e., its timestamp), each box-plot shows the 25 and 75 percentiles, the median (as a flat line), as well as the ending points of the temperature distribution (whiskers). We observe that there is more variation in the temperatures if the time window is longer. This observation is true across all types of nodes. However, closer to the DPR occurrence, the temperature variations decrease significantly for DPR offender nodes as compared to randomly selected nodes and nodes in the same cage as the DPR offender. Unlike previous research for hard-disk related errors [43], our analysis suggests that higher temperature variation does not necessarily increase the probability of DPR errors. In fact, the majority of DPR offender nodes remain comparatively hotter and with non-fluctuating temperatures. We also point out that the temperature variation for randomly selected nodes is also affected by the large number of samples, partially contributing toward higher variance.

Next, Fig. 3.16 presents histograms of the frequencies of temperatures for the three categories of nodes: DPR offenders, DPR cages, and random. The average values are denoted by the dashed lines in each histogram. The figure indicates that for the randomly selected nodes the right tails (corresponding to higher temperatures) are thinner than those of the DPR and DPR-cage ones. Focusing on the histograms that correspond to the 5 min observations (i.e., the upper row of Fig. 3.16), one can notice the difference in shapes across the three histograms. The random nodes, shown in Fig. 3.16(c) have significant probability mass that is below 40 °C comparing to the DPR offenders and non-DPR offenders within the same cage. This mass may not be as pronounced in the 60 min observations, but it is still present across all histograms in the second row of Fig. 3.16. Overall, the six histograms shown in this figure allow the reader to appreciate how mere differences in standard deviation shown in Table 3.2 indeed correspond to significantly



(a) DPR offender nodes (b) DPR offender nodes

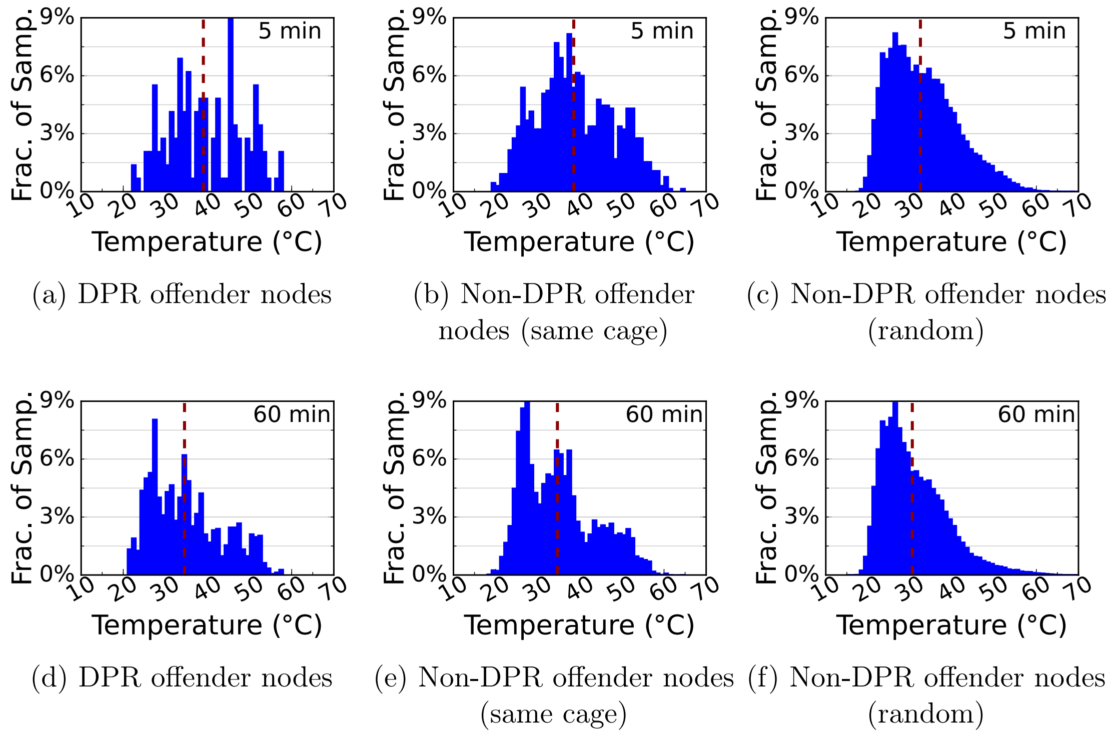
(c) Non-DPR offender nodes (same cage) (d) Non-DPR offender nodes (same cage)

(e) Non-DPR offender nodes (random) (f) Non-DPR offender nodes (random)

Figure 3.15: Temperature variation before each DPR occurrence.

different temperature frequencies.

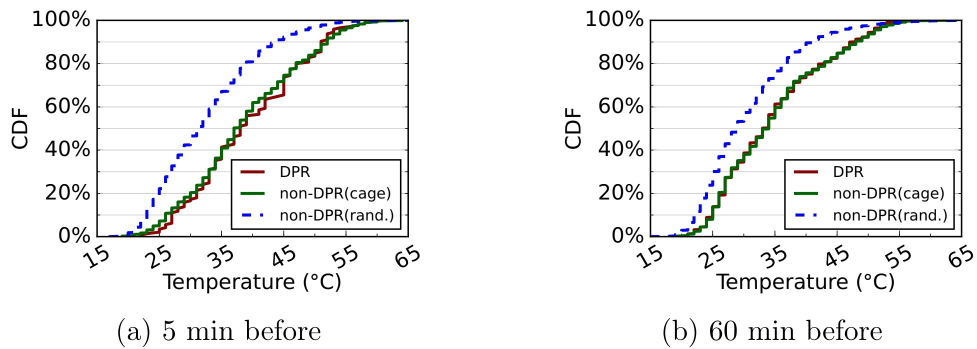
To better compare these histograms quantitatively, we compare them as CDFs in Fig. 3.17. Fig. 3.17(a) shows all CDFs for the 5 minutes case and Fig. 3.17(b) shows all CDFs for the 60 minutes case. Across both graphs, we see that the random nodes (non-DPR) have significantly lower temperature than those of DPR offenders. For example, in the 5-minute window, we see that 50% of random nodes have temperature less than 35°C, but only 25% of those within the DPR case reach this mark. This trend is consistent across most temperatures, nearly 20% of nodes that are randomly selected are consistently cooler than those in the DPR categories (individual and cage). Further we see that even within the same temperature percentile level, there is a difference in temperatures ranging between three to ten degrees. For the longer time window of 60 minutes, these differences still exist but are not as large.



(a) DPR offender nodes (b) Non-DPR offender nodes (same cage) (c) Non-DPR offender nodes (random)

(d) DPR offender nodes (e) Non-DPR offender nodes (same cage) (f) Non-DPR offender nodes (random)

Figure 3.16: Temperature variation before each DPR error.



(a) 5 min before (b) 60 min before

Figure 3.17: CDF of temperature variation before DPR errors.

In summary, we have seen that while the temperatures of DPR offenders may be similar to nodes within the same cage, they are consistently hotter than randomly selected nodes. This further supports the observation that high temperature may precipitate the occurrence of a DPR, especially if it remains consistently high (i.e., temperature variations are rather limited).

We conduct similar analysis for DBE occurrences, results are shown in Table 3.3. We observe that there is no significant difference in temperature of a DBE offender node, other nodes in the same cage as the DBE errors, and randomly selected nodes. Therefore, we can not conclude the effect of temperature on DBEs as per this analysis. However, we found that DBEs occur more frequently in the upper cages than the lower cages (similar to previous work [140]). This indicates some association with temperature, since the upper cages are typically hotter than the lower ones. This in itself can not lead to any well-formed conclusion due the varying temperature of nodes over time. Recall that single bit errors are collected at the start and end of each batch job and hence, we do not have the exact timestamp of occurrence. This limits our capability to perform fine-grained analysis on the effect of temperature on single bit errors.

Table 3.3: Statistics for Temperature (°C) (DBE)

	60min before (avg / stddev)	**15min before** (avg / stddev)	**5min before** (avg / stddev)
DBE	32.64 / 5.97	32.02 / 5.54	33.30 / 6.18
Non-DBE (same cage)	32.14 / 6.24	32.23 / 6.07	33.14 / 6.82
Non-DBE (random)	32.89 / 8.54	32.79 / 7.96	33.39 / 7.89

Observation 3.9 *Temperature may have an impact on GPU soft errors (DPR and DBE), but this conclusion is highly dependent on the choice of nodes to compare against. Our analysis clearly shows that a comprehensive methodology should be followed and described when making such assessments. We find that higher temperatures may be correlated with*

DPR and DBE errors and that higher variabilities in temperature data do not necessarily lead to increased probability for DPR errors.

3.7 Chapter Summary

In this chapter we focus on single bit errors (SBEs), dynamic page retirement errors (DPRs), and double bit errors (DBEs) on the GPUs on the Titan supercomputer and analyze their characteristics and relationships with resource usage, applications, users, and temperature. Our study discovers several previously unknown insights about the characteristics of SBE, DBE, and DPR errors. For example, we show that SBEs happen in bursts and tend to be clustered in time. Average GPU resource utilization and its variance do not seem to be significantly correlated with the SBE occurrences, but shows strong dependence with respect to users and applications. Interestingly, our analysis also shows that top SBE offending GPUs do not necessarily experience more dynamic page retirement errors or DBEs. Another counter-intuitive finding is that SBEs are more likely to occur on the DPR offending GPUs after the DPR error rather than leading to the DPR error. We also provide analysis about possible performance-variation effects of soft-errors and its association with temperature.

Chapter 4

Characterization of Single-Bit Error in the Wild

In the previous chapter, we study the characteristics of three GPU soft-errors on the Titan supercomputer: single-bit errors (SBEs), double-bit errors (DBEs) and dynamic page retirement errors (DPRs). In this chapter, we take a close look on SBEs. We focus on understanding the complex interplay between various kinds of system & workload conditions with SBEs and on discovering insights that are useful for building reliable large-scale systems.

There are several reasons that motivate us to focus on SBEs. First, SBEs occur most frequently comparing to other errors. Their large amount make SBEs an appropriate candidate for analysis on GPU reliability that is of statistical significance. Second, current commercial GPUs support error correction codes (ECCs) on partial components [4, 8, 10]. For instance, a single-error-correction double-error-detection(SDC-DED) algorithm is able to correct SBEs occurred in major memory components, including register files, L1 and L2 caches, shared memory, and DRAM, while parity is capable of detecting SBEs occurred in read-only data cache. In contrast, other structures, such as arithmetic logic units (ALUs), thread schedulers, instruction dispatch unit, and interconnect network, are not protected. Any SBE occurring in those structures would result in incorrect application output or even

program crash. Most importantly, the ECC overhead is significant from the viewpoint of the entire system. On Titan, each K20X GPU contains 6GB of memory. With ECC enabled, the available memory size per GPU reduces by 12.5%, ending in approximately 5.25GB. Given the large number of GPU nodes (18,688) on Titan, a total of 14016GB is used by ECC. In addition to the overhead in storage, ECC also reduces the achievable bandwidth by more than 15% [7]. Therefore, we believe that a study on the characteristics of SBEs is meaningful and would help in discovering opportunities to reduce the ECC overhead, i.e., by dynamically turning on/off ECC.

Before studying the single-bit errors, we first take a look at the characteristics of the temperature distribution on the Titan supercomputer. The motivation is that, temperature has impact on the occurrence of GPU soft-errors (see Chapter 3). Here, we investigate how the GPU temperature distribution varies in time and space across the system. We also compare GPUs with other components, such as CPU and DIMM, and study how their temperature distributions differ from one another over time. In addition, we study how frequently Titan nodes become extremely hot and for how long they stay in such a hot state. We discover that the retention time histogram in the hot state and in the normal state varies significantly between CPUs and GPUs. Interestingly, we find that GPUs switch in and out of the hot and cold states more frequently compared to CPUs and stay in these states for a shorter period of time. We also observe that surprisingly, the retention time of the hot state remains similar for cabinets from different temperature zones.

With a good understanding on the temperature distribution on Titan, we focus on the characteristics of GPU single-bit errors. We show that there exists an interconnection between workload characteristics, certain GPU cards, temperature, power consumption, and GPU single-bit errors. We also show that, it is challenging to exploit this relationship.

This chapter is organized as follows. In Section 4.1, we investigate the temperature behavior on the Titan. In Section 4.2, we explore the relationship between single-bit errors and other related features, such as workloads, GPU node locations, temperature, and power consumption. Section 4.3 gives a brief summary of this chapter. Note that, the

data collection methodology is the same as that described in Chapter 3.

4.1 Temperature Characteristics

Before exploring the relationship between temperature, power consumption, and SBEs, one needs to develop a deep understanding of the temperature behavior on the Titan. Power consumption is highly correlated with temperature, i.e., the Spearman correlation coefficient is as high as 0.5. Consequently, we primarily show temperature analysis. Similar analysis (and conclusions) can also be applied to power consumption. There are two major questions we want to address for the temperature characteristics: (1) what are the GPU temperature characteristics at the node level, and how do these characteristics compare against other components in the system such as CPU and DIMM? (2) what are the GPU temperature characteristics at a coarser granularity such as at the cabinet level, and how do these characteristics differ from CPU and DIMM?

We first show a general view of the temperature on the Titan supercomputer. Figure 4.1(a) presents the empirical pdf of temperature for different components (GPU, CPU, and DIMM) cumulated over the whole sampling period for the entire system. Each Titan node contains one CPU, one GPU, and four DIMMs. Here, we only show the histogram of DIMM A, since all four DIMMs expose similar behavior. We observe that the GPU temperature histogram is fairly spread. While the mean is similar, the variance of the GPU temperature histogram is significantly different from the variance of the CPU and DIMM temperature histograms. This implies that all components attain a range of temperature values over time, and variance may vary from one component to another. Then, we show the monthly empirical pdf for temperature in Figure 4.1(b) and find that the temperature distribution remains steady over time. While the temperature distribution itself may not be used for SBE prediction, such high similarity in temperature distribution over time makes it amenable to learn and exploit it for other purposes by system administrators and facility operators.

Beyond the overall temperature distribution, we are also interested in how frequently Titan nodes become extremely hot and for how long they stay in such a hot state. Prior works [140, 115, 43] suggest that high temperature values are more likely to have high impact on errors. Here we choose *(mean+2std.)* as the hot state threshold. That is, a node enters a *hot state* if its temperature value exceeds *(mean+2std.)*; otherwise, it stays in the *normal state*. Note that the temperature threshold for GPUs ($Thr_GPU{=}40°$) is different from that for CPUs ($Thr_CPU{=}47°$) as the temperature distributions of the two components are different (see Figure 4.1). We define the continuous period during which a node stays in the hot state or normal state as the retention time. Note that we experiment with different temperature thresholds and find that observed trends and insights remain largely similar. We also clarify that these thresholds do not have correlations with utilization levels, i.e., staying in normal state does not imply that the component is idle.

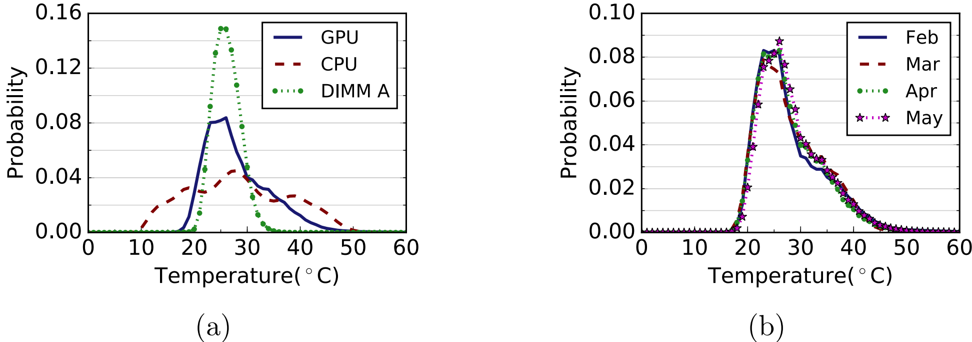
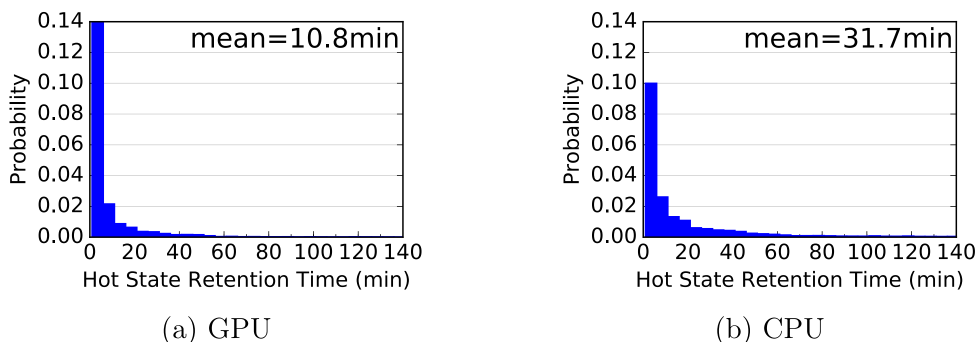


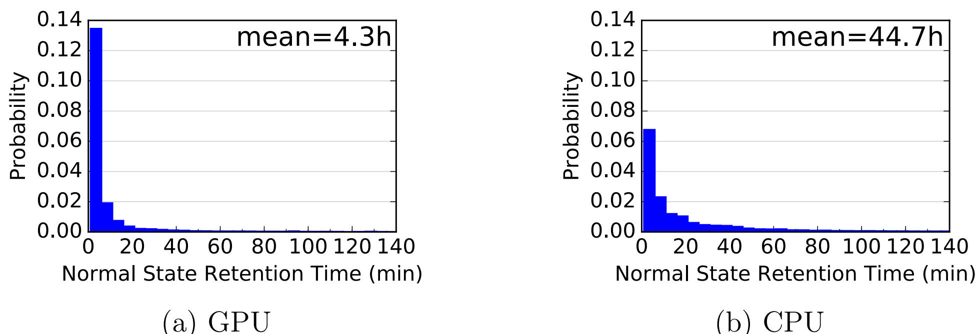
Figure 4.1: (a) Histogram of temperature for GPUs, CPUs, and DIMMs. The average temperature of GPU, CPU and DIMM are 28.1°C, 28.3°C and 26.1°C, respectively. The standard deviation of GPU, CPU and DIMM are 6.1, 9.4 and 2.6, respectively. (b) Monthly Histogram of GPU temperature.

Figures 4.2(a)-(b) show the retention time histogram for the hot state (for GPU and CPU components, respectively), while histograms for the normal state are presented in Figures 4.3(a)-(b). We make several interesting observations. First, the retention time histogram in the hot state and in the normal state varies significantly between CPUs and GPUs, implying a difference in their utilization patterns. Second, GPUs stay in the hot state for a shorter period compared to CPUs (i.e., when GPUs get relatively hotter, they



(a) GPU (b) CPU

Figure 4.2: *hot state*: retention time histogram for (a) GPU and (b) CPU. (Note that the long tail is truncated at 140min in both figures.)



(a) GPU (b) CPU

Figure 4.3: *normal state*: retention time histogram for (a) GPU and (b) CPU. (Note that the long tail is truncated at 140min in both figures.)

are likely to remain in that state for a shorter period compared to CPUs). Recall that the absolute temperature thresholds for entering the hot state are different for GPUs and CPUs. At the same time, GPUs stay in normal state for shorter time too (i.e., when GPUs get relatively colder, they are likely to remain in that state for a shorter period as compared to CPUs). This indicates that GPUs switch in and out of the two states more frequently compared to CPUs and stay in these states for a shorter period of time.

Next, we look at the temperature distribution at the cabinet level (our previous analysis is at the node level). We investigate if the retention time for hot and normal states varies across cabinets with different relative hotness. To achieve this, we first rank all cabinets according to their cumulative temperature over every node. Then, we divide cabinets into three temperature zones based on their cumulative temperature values. We pick the 10

hottest cabinets (*HotCabs*), 10 coldest cabinets (*ColdCabs*), and 10 cabinets ranking in the middle (*MidCabs*), as representatives of the cabinets in each temperature zone. Table 4.1 shows the average retention time of the selected cabinets in each temperature zone for both hot and normal states (for GPU and CPU, respectively).

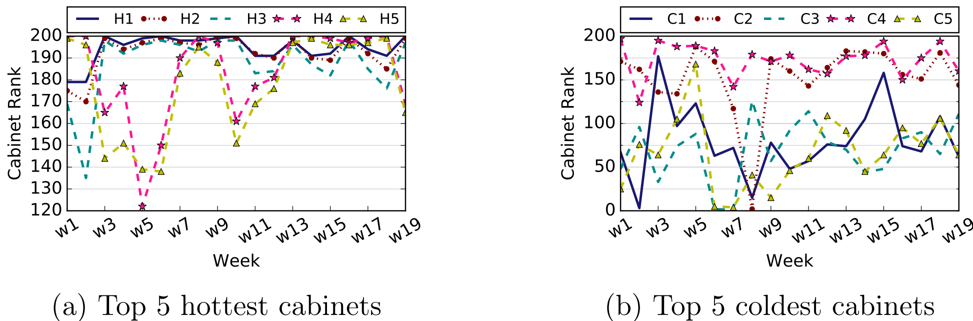
Table 4.1: Temperature mean retention time for cabinets in different temperature zones for GPU and CPU.

Cabinets	**GPU**		**CPU**	
	hot state	**normal state**	**hot state**	**normal state**
HotCabs	10.9min	2.7h	34.6min	22.0h
MidCabs	11.0min	4.2h	31.6min	56.0h
ColdCabs	10.5min	4.9h	31.9min	50.5h

We observe that, surprisingly, the retention time in the hot state remains similar for cabinets from all three temperature zones. Notice that this is not an artifact of the cooling mechanism since it is not a reactive measure that kicks in after a threshold. In contrast, as expected, the normal state retention time of cabinets in *ColdCabs* is significantly greater than that of cabinets in *HotCabs*. In other words, cabinets in *HotCabs* enter a hot state more frequently, this holds for both GPUs and CPUs. While comparing GPUs and CPUs, we notice that the hot/normal state switch is more frequent for GPUs than for CPUs in all three temperature zones (consistent with Figures 4.2 and 4.3).

The preceding analysis is performed over the entire sampling period. It is also interesting to break down the time domain by looking into the dynamic nature of temperature distribution week by week. Towards this goal, we first rank all 200 cabinets according to their cumulative temperature over the entire sampling period. Note that a higher rank indicates a hotter cabinet, i.e., the hottest cabinet ranks 200 while the coldest one ranks 1. We present the weekly ranking over the entire period (19 weeks in total) for the 5 hottest and 5 coldest cabinets, see Figure 4.4. The 5 hottest and 5 coldest cabinets exhibit dramatic variation in their relative hotness ranking over the period. That is, the hot/cold ranking of cabinets changes significantly and frequently over time. We observe similar

trends for other cabinets as well. This observation is particularly interesting because it suggests that although there are hotspot cabinets, these hotspots keep changing over time. Hence, to exploit the correlation between SBEs and temperature for SBE prediction, we must learn and capture this dynamic behavior accurately.



(a) Top 5 hottest cabinets

(b) Top 5 coldest cabinets

Figure 4.4: Weekly ranking for five hottest (a) and five coldest (b) cabinets.

4.2 GPU Error Characterization

Soft errors may occur during an application execution on GPUs for multiple reasons, i.e., cosmic ray strikes, voltage fluctuations, elevated temperature, manufacturing defects, and complex workload-hardware interaction. However, pinpointing the root cause of soft errors is challenging and cannot be easily used to predict soft error occurrences. While soft error occurrences have limited predictability, we find that not all soft error occurrences are random. Our results reveal that certain system and workload properties may have hidden correlations with GPU soft errors, albeit such correlations can not be attributed as causations. In particular, we show that certain GPU cards, workload behavior, GPU temperature, and GPU power consumption may have complex interactions with GPU soft error occurrences.

4.2.1 SBE Offender Nodes

We start by investigating how GPU errors are distributed across the entire system. Since the 200 cabinets on the Titan are organized as a 25×8 grid, we present the normalized

average value of SBE-affected nodes at the cabinet level in Fig. 4.5. Clearly, GPU errors are not uniformly distributed. The number of SBE-affected GPU cards are not the majority of all cards in the system either. Exploiting this observation in isolation is not likely to yield good prediction of future SBEs. For example, if we predict that all applications executing on these SBE offender nodes will experience errors, it results in a high false positive rate because SBE offender nodes do not experience errors uniformly over all days either. Actually, 80% of error offender nodes experience a soft error on less than 20% of the total days over the trace period. Nevertheless, the non-uniform distribution of soft error offender nodes in the space and time domains open the possibility for learning-based predictions.

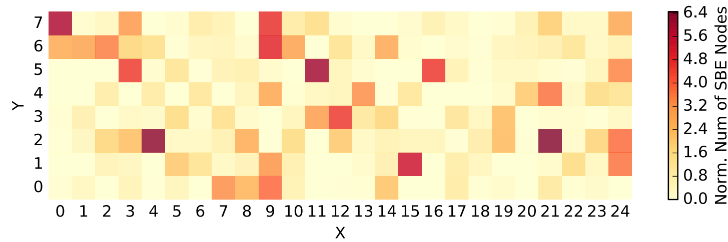


Figure 4.5: Non-uniform distribution of GPU error offender nodes at the cabinet level.

4.2.2 Application

It is also important to analyze the impact of various workloads on GPU soft error occurrences. As a first step, we explore the spatial distribution of SBE-affected applications and observe the non-uniform distribution across the Titan system (see Fig. 4.6).

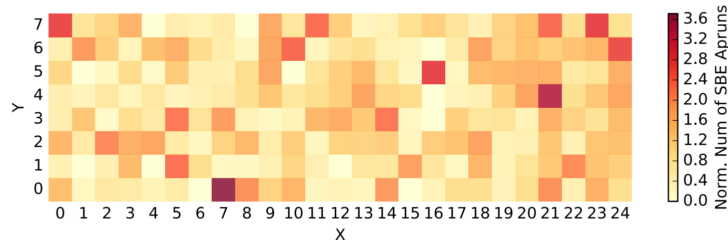
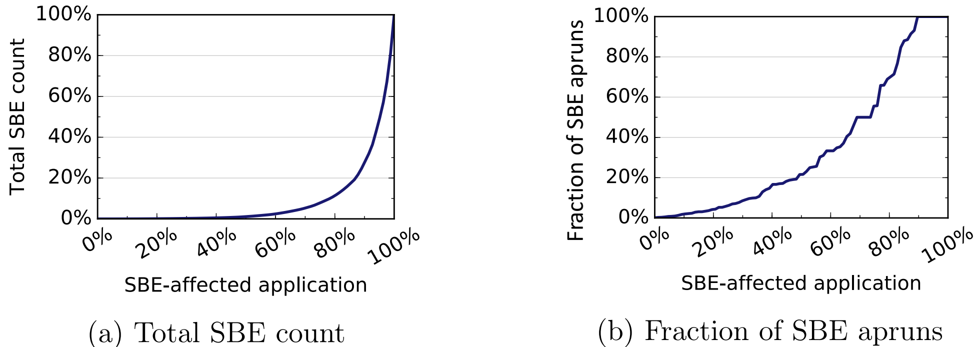


Figure 4.6: Non-uniform distribution of SBE-affected application runs at the cabinet level.

Next, we look at the severity of SBE-affected applications by analyzing their SBE count (application and SBE correlation is normalized by the GPU core hours, i.e., runtime × number of nodes). Fig. 4.7(a) shows that a smaller set of workloads, less than 20% of all applications, experience the majority of errors (≥ 90%).



(a) Total SBE count (b) Fraction of SBE apruns

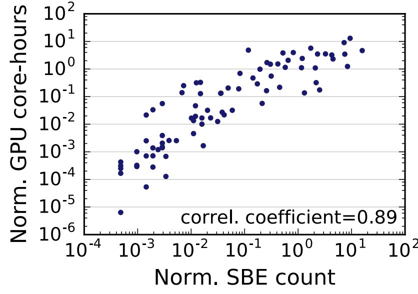
Figure 4.7: Workload and GPU error distribution: a small set of workloads experience most of the soft errors (a), and fraction of executions affected by SBEs for SBE-affected application runs (b).

Fig. 4.7(b) shows that even SBE-affected applications do not experience SBEs uniformly across all application runs. The top 20% of the SBE affected workloads experience all their share of soft errors during 60% of their total application executions, while the lower 20% of the SBE affected workloads experience all their share of soft errors during less than 10% of their total application executions.

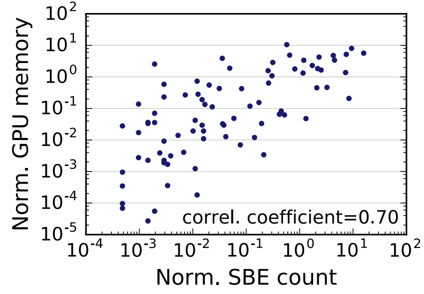
We further investigate the relationship between the severity of SBE-affected application runs and their GPU utilization, i.e., core-hours and memory, see Fig. 4.8. The high Spearman coefficient values (see inset in each figure for the exact values) indicate that applications with more SBEs tend to utilize more GPU memory and for longer duration. The above observations imply that application related measurements such as utilization, are good indicators for SBE occurrences.

4.2.3 Temperature and Power Consumption

Consistent with previous studies on GPU errors [55, 101, 140, 102], we analyze the potential relationship between GPU temperature/power consumption and GPU errors.



(a) GPU core-hours

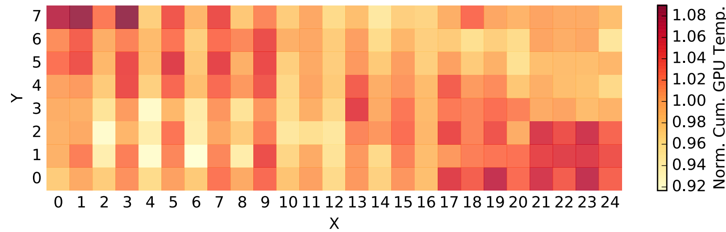


(b) GPU memory

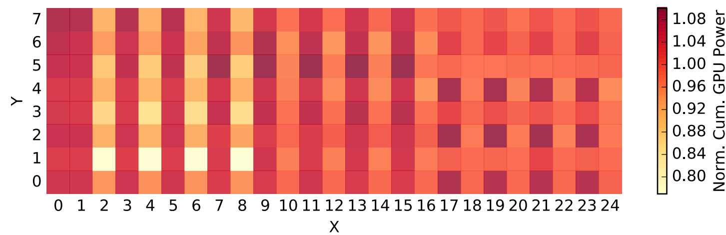
Figure 4.8: Scatter plot of SBE count of SBE-affected application runs and their GPU utilization: core-hours (a) and memory (b).

4.2.3.1 A bird's eye view

We first explore whether GPU temperature/power consumption correlate with soft error occurrences. Fig. 4.9 shows the cumulative temperature/power consumption over the entire sampling period of every cabinet in the Titan.



(a) Temperature distribution



(b) Power consumption distribution

Figure 4.9: Distribution of temperature (a) and power consumption (b) accumulative over the whole period at the cabinet level.

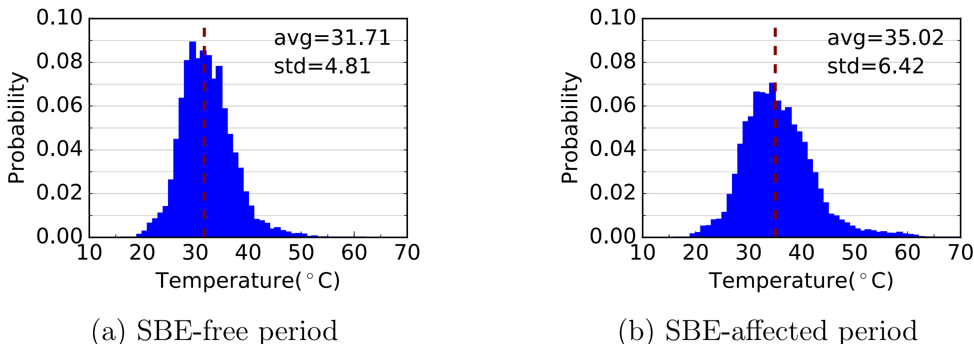
We observe that the temperature distribution is non-uniform in space, i.e., cabinets in the upper left corner and lower right corner tend to be hotter than the rest. In contrast,

power consumption is more evenly spread, implying that the Titan is intensively utilized both time-wise and space-wise.

Next, we compare the non-uniform temperature distribution with the SBE-affected node distribution (Fig. 4.5) by calculating the Spearman correlation coefficient at the node level. The low value (0.07) implies that the accumulative temperature distribution is not related to the distribution of SBE offender nodes in space. The same observation is reached when comparing the temperature distribution and the SBE-affected application distribution (the Spearman correlation coefficient is only 0.15). Similar analysis is conducted for power consumption, which also shows weak correlation between power consumption and SBE-affected nodes or SBE-affected application runs. In summary, the effect of temperature on SBEs may not be entirely captured by SBE offender nodes or workload characteristics only.

4.2.3.2 Considering the time dimension

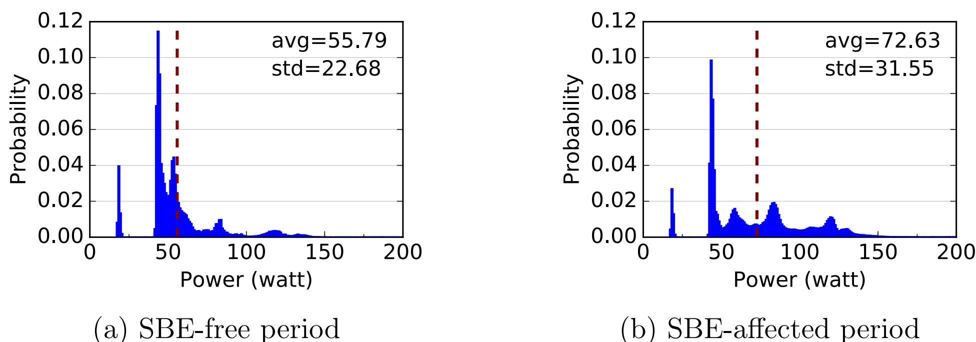
We turn the focus to SBE offender nodes and temperature characteristics across time. We divide the time dimension in two parts: (1) the time during which a soft error occurs (*SBE-affected period*) and (2) the time during which no soft error occurs (*SBE-free period*). Fig. 4.10 shows the empirical temperature distribution of SBE offender nodes during these two periods. The distribution for GPU power consumption is presented in Fig. 4.11.



(a) SBE-free period

(b) SBE-affected period

Figure 4.10: Temperature distribution of SBE offender nodes during SBE-free periods (a) and SBE-affected periods (b). Vertical lines represent mean values.



(a) SBE-free period (b) SBE-affected period

Figure 4.11: Power consumption distribution of SBE offender nodes during SBE-free periods (a) and SBE-affected periods (b). Vertical lines represent mean values.

We observe that the SBE offender nodes are relatively hotter during the SBE-affected period by more than 3°C on average, compared to SBE-free period (Fig. 4.10(a) vs. 4.10(b)). The SBE offender nodes also consume relatively higher power during the SBE-affected period by more than 15 watts on average, compared to the SBE-free period (Fig. 4.11(a) vs. 4.11(b)). Note that higher power consumption likely contributes to increased temperature. However, due to varying cooling efficiency and workload characteristics, temperature elevation may be caused by other factors too. The above observation implies that SBEs are more likely to happen during periods of elevated temperature. Our measured data do not conclusively indicate that SBEs definitely occur above a certain threshold of temperature/power consumption. Sometimes even during the SBE-free period, temperature can be significantly high (see Fig. 4.10(a)), making the relationship between SBE occurrence and temperature/power consumption not straightforward.

We caution that the effect of temperature or power consumption on SBEs is still not conclusive. The preceding analysis only considers SBE offender nodes – providing limited view of the whole system. For example, our previous analysis does not show that non-SBE offender nodes consistently attain lower temperature than SBE offender nodes during the SBE-affected period. So temperature and power consumption characteristics of non-SBE offender nodes should also be considered. Unfortunately, performing a meaningful and accurate data analysis on non-SBE offender nodes is challenging for multiple reasons.

First, the number of non-SBE offender nodes is large ($\geq 17,000$ nodes) as compared to SBE-offender nodes ($\leq$ 700 nodes). Second, the long observation period of this study induces difficulties in analyzing temperature and power consumption data in a meaningful and representative manner.

An intuitive solution to this problem is to randomly sample a subset of non-SBE offender nodes and perform comparisons with SBE-offender nodes. Unfortunately, this method leads to inaccurate conclusions. Random sampling of non-SBE offender nodes may include idle time at certain GPUs and hence, may likely result in lower average temperature and power consumption values. An alternative method is to sample only active GPUs at a given time. However, we found two issues that impede the practicality of this solution. First, current GPU resource utilization monitoring tools can not be used at runtime to monitor GPU utilization without imposing significant overhead on production systems. Second, sampled GPUs can execute workloads that finish at different times than the SBE-affected period on SBE offender nodes. To mitigate these challenges, we find that comparing against the other nodes in the same cage for a given SBE offender node result in consistent comparisons. The reasons are: (1) the nodes in the same cage are likely to be active at the same time due to the scheduling policy which is likely to pack one job in the same cage, (2) nodes in the same cage are likely to show similar variation in temperature due to power/cooling and spatial locality.

Table 4.2 shows the mean and standard deviation of the temperature and power consumption for non-SBE offenders in the same cage during SBE-affected and SBE-free period as observed on the SBE offender node (see Figure 4.10 and 4.11). We observe that even non-SBE offender nodes are relatively hotter during SBE-affected period compared to SBE-free period. Note that non-SBE offender nodes do not experience any SBE during an SBE-affected period or an SBE-free period. In addition, while non-SBE offender nodes are relatively hotter during the SBE-affected period, the SBE offender node is on average hotter than non-SBE offender nodes in the same cage. Similar observations can be drawn for power consumption. The above observations imply that temperature and power con-

sumption may have some effect on SBE occurrence, but, it is challenging to quantify the correlation due to monitoring limitations and interaction of other possible factors.

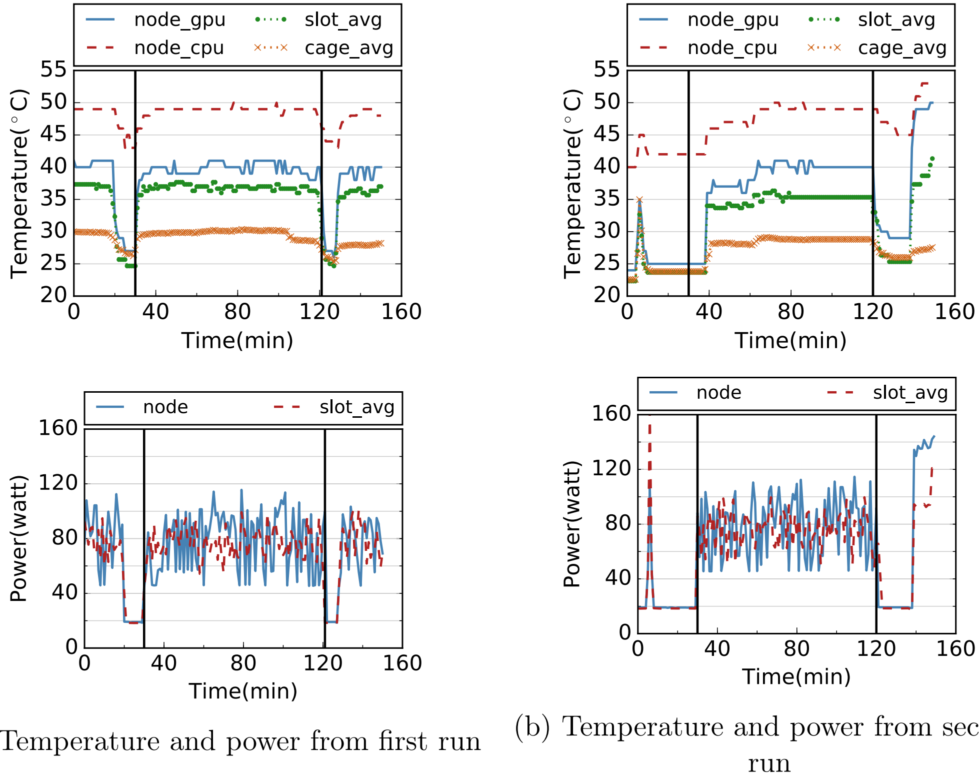
Table 4.2: Statistics of temperature and power on Non-SBE offenders.

Time Period	Temperature (°C)		Power (watt)	
	Mean	**Std.**	**Mean**	**Std.**
SBE-affected Time	34.30	6.76	68.22	33.09
SBE-free Time	30.44	5.21	48.17	23.79

4.2.3.3 Considering the space dimension

Besides the time domain, it is natural to also explore whether similarities exist across space. In fact, our measured data indicate that GPU power consumption and temperature profile can change for the same workload across runs, possibly due to effects from neighboring nodes (i.e., spatial effects). We first investigate how the temperature profile changes when the same workload is executed repeatedly on the same node. Intuitively, one does not expect the temperature profile to change. To test this, we select a computational chemistry application that is executed multiple times on the same node at different times. Fig. 4.12 shows the temperature and power profiles of GPU during two different runs on different days, but on the same node to avoid location specific power/cooling side-effects. We plot the average temperature and power values for all other nodes in the same slot or cage, as well as the temperature profile of the CPU in the same target node. For the power profile, we do not have the ability to measure CPU power consumption out-of-the-band. We include the 30 min time window before and after the application run to evaluate the results in context.

From Fig. 4.12, we observe that the temperature profile changes from one run to another and that it is not necessarily correlated to fluctuations in the power profile. The graph indicates that changes in the temperature/power consumption of neighboring nodes and the CPU in the same target node may contribute to the variation in the temperature



(a) Temperature and power from first run

(b) Temperature and power from second run

Figure 4.12: Effect of neighboring components on temperature/power of an application over two runs on the same node overtime. Vertical solid lines represent the start and end of the aprun execution.

profile of the target node. Other factors such as change in power/cooling efficiency in the spatial region may also contribute to variation in the temperature profile, although these factors are hard to detect and quantify. Motivated by the above evidence, we argue that temperature and power consumption from neighboring nodes in the same slot, as well as the temperature of the CPU on the same node, may also help with SBE occurrence prediction. Still, it is non-trivial to understand whether or how much the behavior of neighboring nodes can actually improve error prediction capabilities.

4.3 Chapter Summary

In this chapter, we take a close look into the single-bit errors on the Titan's GPU nodes. We reveal several interesting and useful insights obtained via studying the complex impact

of certain GPU cards, workloads, temperature, and power consumption on the occurrences of GPU single-bit errors. These insights indicate that these factors could have predictive or associative capabilities with GPU errors, but it is non-trivial to exploit them to guide the design of low-overhead yet reliable GPU-accelerate systems. In Chapter 5 and 6, we demonstrate that simple schemes based on these observations show poor indicative capability of GPU errors. On the other hand, machine-learning-based approaches, which are able to capture the hidden interactions between GPU soft-errors and their related features, are able to accurately predict the error occurrences. Such predictability is useful in building reliable systems with lower overhead, such as systems that allow to dynamically turning on/off ECC protection based on prediction results. This saves the protection overhead when the probability of SBE occurrence is very low.

Chapter 5

Predicting GPU Soft-Errors with Neural Networks

In the previous chapter, we have discovered that workload characteristics, certain GPU cards, temperature, and power consumption could be indicative of GPU single-bit errors. However, it is non-trivial to naively exploit them for error prediction. Motivated by these observations and challenges, in this chapter, we resort to machine-learning models to capture the hidden interactions between GPU SBEs and their related factors. Such models are useful in guiding flexible error protection schemes for GPU nodes, e.g., by dynamically turning on/off error protection based on prediction results.

One may argue that completely turning off error protection may be too risky. However, it is important to notice that the impact of error-correcting code (ECC) overhead on real-world computational science applications can be as high as 10% on GPUs [26]. Specifically for the Titan supercomputer, we have pointed out in Chapter 4 that enabling ECC reduces the available memory size per GPU node by 12.5% and decreases the achievable system memory bandwidth by more than 15% [7]. Nevertheless, the decreased memory bandwidth caused by ECC overhead can result in larger performance degradation than the decreased fraction of bandwidth itself due to queuing. In fact, computational scientists already naively turn off ECC for their application runs [53]. In such cases, a prediction model

would be useful instead of always turning off ECC.

Acknowledging the necessity of an error predictor, in this chapter, we elaborate on the challenges, process, and solutions involved in building effective machine-learning-based prediction models. The goal of the predictor is that given an upcoming application to be launched on a certain GPU node, to determine whether this application is going to encounter any SBE occurrence or not (i.e., a binary classifier). Here, we choose artificial neural networks (NNs) as the model because of its superior ability in capturing the non-linear functions between the input features and the prediction target [21]. This chapter makes the following contributions:

- We select a host of factors, including resource utilization, node location, application type, temperature, and power consumption, as input features for prediction models.
- We discover the presence of imbalanced dataset challenge in the Titan dataset. In addition, we overcome the challenge with a customized algorithm, which is based on similarity comparison of the feature sets among different samples in the majority class.
- We propose a neural-network-based predictor that is able to accurately forecast the SBE occurrences.

This chapter is organized as follows. We summarize related work in Section 5.1. Section 5.2 briefly introduces the steps required in building machine-learning-based prediction models. In Section 5.3, we elaborate on the methodology and challenges in designing the neural-network-based SBE occurrence prediction model. The proposed model is evaluated in Section 5.4. In Section 5.5, we discuss several open problems and challenges. Section 5.6 provides a brief chapter summary.

5.1 Related Work

5.1.1 Applications of Machine Learning Models in Systems

There is a rich body of prior works that utilize machine learning models to predict performance measures in systems and data centers. Couceiro et al. [36] leverage neural network to predict performance of the total order broadcast in fault-tolerant replicated systems. Nikravesh et al. [105] and Hu et al. [61] resort to support vector machine to predict the workload pattern in auto-scaling systems in the Infrastructure as a Service (IaaS) layer of cloud computing and for grid resource monitoring and prediction, respectively. Other works [22, 38, 40] take advantage of other machine learning models, such as Bayesian models and decision tree, to predict system workload.

In addition to system workload and performance prediction, machine learning models also help in forecasting system failures. For example, researchers leverage various kinds of machine learning models to predict drive failures in order to improve the reliability of storage systems [91, 87, 99]. Zhang et al. [157] use support vector machines for switch failure prediction in data center networks.

In this chapter, we show the effectiveness of using neural networks for predicting the occurrences of SBEs on Titan's GPU nodes. In Chapter 6, we apply more machine learning models (i.e., logistic regression, support vector machine, and gradient boosting decision tree) and make comparisons in terms of the prediction quality and model overhead across various models.

In this work, we use neural networks to successfully predict the number of SBE occurrences at the node level and at the cabinet level in a large-scale HPC system. The use of neural networks is necessary for predicting SBE occurrences as the statistical analysis that has been used for prediction [55] is insufficient here. The proposed neural network combines a set of features that can be used as a whole for SBE prediction and shows that in addition to node location, utilization and workload type, temperature is also important for future SBE prediction.

5.1.2 Time Series Prediction

When building the machine-learning model, some of the selected input features cannot be known prior to application execution, such as the temperature and power consumption during the application run. Therefore, we need to leverage time-series prediction tools to forecast those features. Fortunately, there are plenty prior works focusing on this purpose. Time series prediction tools (i.e., ARMA/ARIMA [28] and Holt-Winters exponential smoothing [52]) have been widely applied to quantify the impact of workload changes to application and/or system performance [142, 158, 130, 152]. Tran et al. [142] leverage ARIMA to improve block prefetching for scientific applications while Zhuang et al. [158] use ARIMA for effective user traffic prediction for capacity planning. Compared to traditional models, neural networks have shown to be efficient in capturing irregular patterns in data center resource usage [151, 150, 149], effective characterization of TCP/IP [35], and web server views [86].

Here, we leverage PRACTISE [151] to first forecast features based on time series and then plug them into the proposed models for SBE occurrence prediction. We choose PRACTISE over other solutions because of its capability of accurately capturing short-term (i.e., as short as 15 minutes) temporal dependency in the time series, which fits the purpose of predicting features before the execution of applications.

5.2 Overview of the Methodology

In this section, we describe the general procedure of building a machine-learning-based predictor and discuss the several commonly-used evaluation metrics that are able to quantify the prediction quality of machine learning models.

In Chapter 4, we illustrate that GPU errors are potentially correlated with different system and workload characteristics. Formally, we are interested in finding a mathematical function that maps these properties (features) to the probability of GPU error occurrence. If we express system and workload dependent properties as features $x_0, x_1, x_2, \ldots, x_n$, there

exists a function F_{pred}, such that the probability of GPU error occurrence during program execution is expressed by:

$$Prob_{err} = F_{pred}(x_0, x_1, x_2, \ldots, x_n) \; . \tag{5.1}$$

Note that, many such functions can exist with varying accuracy-levels because the probability of GPU error occurrence during program execution may not always be dependent on the value of different features only. It is possible that a mathematical function can not fully capture the behavior because of the inherent randomness involved with soft error occurrences. Therefore, the goal is to "learn" a classification function, F_{pred}, that provides high accuracy based on the available features. Given this, we take the following steps:

Step 1: Feature selection and engineering. We select a set of features as input to the desired function. We elaborate the process, challenges, and solutions involved in selecting a useful set of features (Section 5.3.1).

Step 2: Function discovery. We discuss how to learn the desired classification function in a generic yet meaningful way. We provide details on the challenges in learning the classification function (Section 5.3.3).

Step 3: Analysis of the learned function. We investigate the usefulness of the learned function and analyze the function to assess if it can provide meaningful results under different circumstances (Section 5.4).

We emphasize that these steps are means to show that such a problem can be solved with reasonable accuracy and under practical constraints.

Evaluation metrics selection: In order to perform **Step 3**, it is important to choose meaningful metrics to measure the goodness of the machine learning model. Accuracy is a simple and widely used metric to assess the effectiveness of predictions. However, it is misleading for evaluating imbalanced datasets (which is present in our dataset, see Section 5.3.2). A naive method, such as always predicting every sample as the majority

class, can lead to high accuracy for highly imbalanced dataset but fails to predict the minority class. Here, we leverage three commonly-used evaluation metrics for classification tasks, including precision, recall, and F1 Score [118].

Precision is defined as the ratio of correct predictions (true positives) to all predictions (true positives and false positives):

$$Precision = \frac{True\ Positives}{True\ Positives + False\ Positives}\ , \tag{5.2}$$

while **Recall** reveals the ratio of identified samples to the ground truth, expressed by the following formula:

$$Recall = \frac{True\ Positives}{True\ Positives + False\ Negatives}\ . \tag{5.3}$$

The value of both precision and recall falls in the range between 0 and 1. The higher the value, the better the prediction quality. The main goal of any prediction mechanism is to improve both precision and recall. However, precision and recall sometimes can be conflicting, i.e., increasing precision might result in decreasing recall and vice versa. This is because as we increase the true positives, the false positives may also increase [29]. Consequently, we resort to **F1 Score**, the harmonic mean of precision and recall (see Eq. 5.4),

$$F1\ Score = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{5.4}$$

as another evaluation metric to additionally capture the trade-off between prediction and recall. A higher F1 score indicates better prediction quality.

In this chapter, we only utilize precision and recall for evaluation as we only use one machine learning model (that is, neural network) here. In Chapter 6, we use the F1 score as we apply four different machine learning models and F1 score allows for easier comparisons across models.

5.3 SBE Prediction Framework

Our characterization results reveal a relationship among temperature, power, and SBE occurrences, but not a clear one. It is unclear how to accurately predict SBE occurrences simply based on the statistical properties of temperature and power. In this section, we resort to machine learning models to explore whether the time series of temperature, power, and other features can be used to predict SBE occurrences.

5.3.1 Feature Selection

To begin with, we discuss how to select features that are potentially related to an SBE occurrence. Determining an effective set of features to learn the desired function is challenging. First, measuring and collecting plausible features correlated with GPU errors is not always possible. For example, the memory access pattern could be associated with SBEs. However, the overhead to collect this information in a production system with dynamically changing workloads is cost prohibitive. Second, selecting features from what can be measured and collected is taxing. One can conservatively collect data from all instrumentation sources, but it may result in excessive storage and processing overhead without clear understanding if they are indeed related to the final outcome. Consequently, feature selection is a critical aspect toward learning the desired function. We refer to the process of transforming the selected features into quantifiable and meaningful representation as feature engineering.

Our feature engineering is guided by the characterization analysis in Chapter 4, where we observe that workload characteristics, certain GPU cards, temperature, and power consumption could be indicative of GPU SBEs. The following list shows the selected input features:

- **Temperature:** We use the mean and standard deviation of temperature during an application run as input features. To account for dynamic temperature behavior, the

mean and standard deviation of the temperature *difference* between two consecutive minutes are also selected.

- **Power:** Similar to temperature, four metrics are selected for power: mean and standard deviation of consumed power during the application run, and mean and standard deviation of the power difference between two consecutive power measurements.

- **Node location:** Row, column, and cage indices for each node are included (recall that the Titan is organized as a two-dimensional grid of cabinets, with each cabinet consisting of three cages).

- **Memory utilization:** GPU memory utilization for every node that the application is assigned to.

- **Application:** The application run execution time and the application vector are also considered as features. The walltime of the application run is the value normalized by the total number of nodes launched by this application run, while the application vector represents which application is executed.

5.3.2 Challenge: Imbalanced Data Set

After discussing the feature selection process, we provide details on the training data set. We collect data for all application runs during the sampling period. For application runs executing on SBE offender nodes, we divide the node's busy time into two parts: (1) SBE-affected time, if the application run sees at least one SBE; otherwise, (2) SBE-free time. Busy time is defined as the time when a given GPU node is not idle. By definition, for non-SBE nodes, the busy time is always SBE-free time. We use the first three and half months of our entire sampling data as the training data set, this encompasses about $70,000$ application runs (i.e., 6 million samples). Each sample is identified by $<applicationrun_id, node_id>$. For example, an application run launching on 5 nodes will produce 5 samples. Note that the number of application runs per month are not the

same across each month. We select the first three and half months to collect enough observation samples. Indeed, as shown later in Section 5.4.1, the testing data set contains the samples in the following two weeks, which encompass 16,000 application runs, such that the testing data is about 23% of the size of the training data, which is around the rule-of-thumb ratio of the testing data set to the training data set [56].

Our first effort is to use the raw samples as input to train a machine learning model, e.g., a neural network. Unfortunately, both precision and recall for SBE occurrence is as low as 0.01 while the precision and recall for non-SBE occurrence prediction is as high as 0.95. Clearly, the low precision and recall values for SBE occurrences imply that such a naive model is not useful as it mislabels all samples as non-SBE.

Looking into the training data set, we find that the raw training data set is extremely imbalanced: almost 98% of all training data are non-SBE occurrence samples, which results in a highly biased model. Imbalanced data sets is a noteworthy difficulty to machine learning models [120] as the resulted models favor the majority class and almost ignore the minority class, which is exactly what we observe here. To mitigate the imbalanced data problem, there are two common solutions [120]: (1) over-sample the minority class or (2) under-sample the majority class. Over-sampling replicates samples by creating synthetic minority samples based on nearest neighbors [30]. Here, we opt for under-sampling of the majority class, since this allows us to work with real rather than synthetic data.

One method for under-sampling is to *reduce similar samples* in the majority class. Here, we propose a customized under-sampling method, which is based on similarity comparison of the feature sets among different training samples from the same application run in the majority class. Algorithm 1 shows how to select representative samples for one application run. The key idea is that if two feature sets are highly correlated, we only select one of them for training. The algorithm inputs are: 1) the normalized features of all training samples for this application run, denoted by S, and 2) a threshold ρ_{thres}, used to determine whether the Pearson correlation of the feature sets is strong enough. The larger the ρ_{thres}, the stronger the similarity between the samples. Sample thinning is based on ρ_{thres} as

Algorithm 1 Select representative samples for one application run based on feature correlation.

```
1: procedure SIMILARITY_REDUCTION(S, ρ_thres)
2:     // Get each sample's high correlated samples
3:     high_corr_samples ← hashtable(sid, {});
4:     for s_i in S do
5:         feature_i ← feature list of sample s_i;
6:         for s_j in S do
7:             feature_j ← feature list of sample s_j;
8:             corr ← pearson_corr(feature_i, feature_j);
9:             if corr ≥ ρ_thres then
10:                high_corr_samples(s_i) ← s_j;
11:            end if
12:        end for
13:    end for
14:
15:    // Sort in descending order
16:    Sorted_S ← sort(size(high_corr_samples(sid)));
17:
18:    // Select representative samples for this application run
19:    selected ← {};
20:    avail ← Sorted_S;
21:    for s_i in Sorted_S do
22:        if size(avail) ≠ 0 then
23:            selected ← s_i;
24:            avail.remove(s_i);
25:            for s_j in high_corr_samples(s_i) do
26:                avail.remove(s_j);
27:            end for
28:        end if
29:    end for
30:    return selected;
31: end procedure
```

smaller values for ρ_{thres} force more aggressive data reduction. We repeat the algorithm for every application run in the training data set for the majority class. Several correlation threshold values for ρ_{thres} can be used. With ρ_{thres} less than 0.7, the data set reduction is too aggressive and too few representatives are left for the majority class when compared with the original minority class. This confirms our assumption that there are plenty of redundant training samples in the majority class. We select $\rho_{thres} = 0.9$ as the threshold value, this selection reduces the raw data sufficiently.

Note that $\rho_{thres} = 0.9$ is a choice that achieves good reduction of the dataset but

certainly not the only one that can be used. Experimentation shows that various ρ_{thres} values close to 0.9 are also effective. Moreover, to avoid favoring some application runs (we certainly want to avoid an imbalanced data set for application runs), we guarantee that for each application run we select at least 2 training samples. The above efforts result in a significant data set reduction to a total of 0.2 million samples of which 60% are non-SBE occurrences and 40% are SBEs.

5.3.3 Model Selection

In this chapter, we leverage a neural network model to predict SBE occurrences due to its superiority in capturing the complex interactions between input features and the prediction target [21]. Artificial neural networks are inspired by biological neural networks and are composed of many interconnected neurons [21]. The weights associated with the neurons are used to approximate non-linear functions of the input features and are tuned during training. Our purpose here is to use neural networks to explore hidden relationships among the selected features (i.e., temperature, power consumption, and utilization) and upcoming SBE occurrences. Note that, building the training dataset and evaluating the trained model is an iterative process that aims to refine the learned classification function as time passes. Here, the model construction is relatively less frequent (i.e., once every two weeks).

5.4 Evaluation

In this section, we evaluate the proposed neural-network-based prediction model. Note that, some of the selected input features (see Section 5.3.1 for feature selection) for the target application run can be collected prior to execution (i.e., node location and application information), while certain program specific features such as GPU power and temperature profiles can not always be known a priori. We experiment with two approaches and achieve similar results.

- In the first approach, the prediction can be done at the end of the application execution, and a possible re-execution may be required depending on the program's resilience needs. In this case, all input features are known correctly.

- The second approach is that certain input features are learned using statistical models and are fed into the learned function. Note that, this approach can not guarantee that all input feature values are 100% accurate. Fortunately, HPC workloads are fairly repetitive. It is possible to effectively learn and accurately predict program specific features, i.e., their temperature and power profile, by leveraging time-series prediction tools, e.g., [28, 151].

Here, we leverage PRACTISE [151], a neural-network-based time-series prediction tool. In Section 5.4.2, we explain the reason that we choose PRACTISE over other solutions and show the effectiveness of PRACTISE capturing the feature time series. We show that the SBE predictor using the values partially inferred by PRACTISE (see Section 5.4.3) is able to achieve the same quality as the one using oracle values (see Section 5.4.1).

5.4.1 Evaluation with Oracle Data

For the testing data set, we choose two weeks after the training period (2015/5/16-2015/5/29), containing $16,000$ application runs, i.e., 0.5 million samples, bringing the ratio of application runs of testing versus training to 23%. Here, we assume that we know a priori for features that cannot be known before application execution, such as temperature, power, and utilization, to test the neural network model. In the next subsection, we will discuss how to predict these features in advance and compare with the results shown here.

Table 5.1 shows the precision and recall of non-SBE and SBE occurrences for the testing data set using *three* different neural networks: one with all features described in Section 5.3, one with all features except power, and one with all features except temperature. All three models have similar prediction quality, while the one without power is slightly better than

the rest two. Precision and recall are higher than 0.69 for all three cases, suggesting that all models can identify most of SBE occurrences. Besides, we notice that the model without power and the one without temperature expose similar prediction ability. This is understandable since temperature and power consumption are highly correlated as stated previously. In the remaining of this section we focus on the neural network model that is trained without the power data. Note that we conduct experiments with all three neural networks and results are indeed very close to each other.

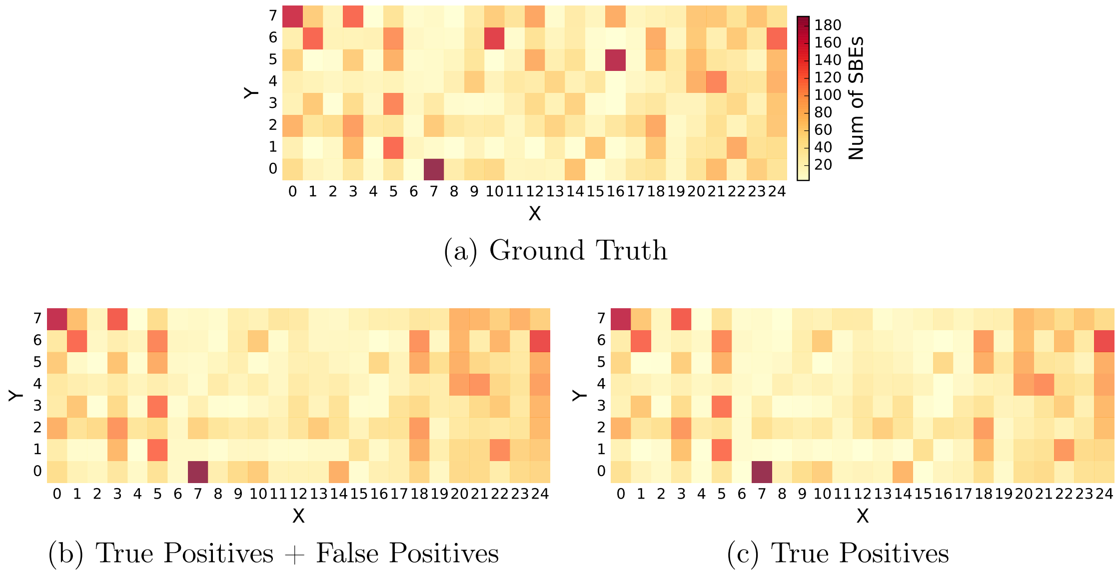
Table 5.1: Precision and recall for three neural networks.

Feature Set	**Non-SBE**		**SBE**	
	Precision	**Recall**	**Precision**	**Recall**
All Features	0.76	0.70	0.71	0.78
No Power	0.78	0.69	0.71	0.80
No Temperature	0.77	0.69	0.70	0.78

Precision and recall give an overview of the goodness of prediction. Figure 5.1 shows a fine-grained SBE prediction quality at the cabinet level throughout the Titan layout. Recall that, the cabinets on Titan are organized as a 25×8 grid. In the heatmap, each cell represents the number of SBE occurrences per cabinet. A color closer to red indicates there are more SBE occurrences in that cabinet. Figure 5.1(a) corresponds to the ground truth. Figure 5.1(b) presents all SBE predictions at the cabinet level, including those correct predictions (true positives) and incorrect predictions (false positives). In contrast, Figure 5.1(c) only shows the correct predictions (true positives). For the majority of the cabinets, SBE prediction is quite close to ground truth with the exception of the middle upper part in Titan's layout.

To deliver a better overview of prediction, we compare the cumulative distribution plots of SBEs across the entire system to the ground truth, all predictions (true positives plus false positives), and true positives, see Figure 5.2. The three CDFs are close to each other, which further confirms that the neural network prediction is overall successful.

In addition, we observe that there are around 5% of cabinets where the neural network



(a) Ground Truth

(b) True Positives + False Positives

(c) True Positives

Figure 5.1: SBE occurrence prediction at the cabinet level.

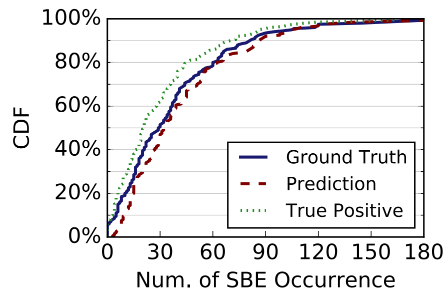


Figure 5.2: Comparison between CDFs of ground truth, all prediction, and true positives for SBE occurrences at the cabinet level.

underestimates SBEs. These cabinets correspond to the ones in the upper middle part ($9 \leq X \leq 16$ and $5 \leq Y \leq 7$) of the cabinet layout, see Figure 5.1. To better understand why the neural network sometimes fails, we focus on two cabinets with underestimates and two cabinets with good predictions. For each SBE occurrence sample in the testing data set, we compute the correlation of feature sets, with every SBE sample and non-SBE sample in the training data set. We find that in the two cabinets with poor prediction 59% SBE occurrence samples in the testing data set have similar features to non-SBE samples in the training data set. This number is dramatically low (only 5%) for the cabinets where the prediction is good. Essentially, it is not possible for the neural network to perform well

for the cabinets with such close feature similarities. Perhaps more features for training are needed to increase prediction robustness to cover such situations.

5.4.2 PRACTISE for Feature Prediction

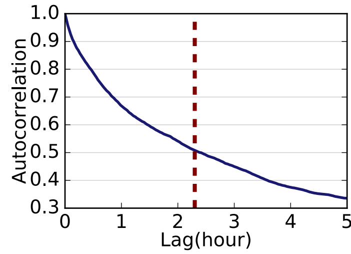
In order to predict future SBE occurrence, we need to *predict the input features*. Location, memory utilization, and application related features are constant overtime, thus we use the average of recent observations as input. Temperature and power are not constant but rather fluctuate across time. To solve the challenge of temperature/power prediction, we leverage PRACTISE [151], which is a neural network prediction tool for time series data that is publicly available.

For PRACTISE to be successful, the time series needs to show temporal dependency. We use autocorrelation to quantify temporal dependency [82]. Figure 5.3(a) shows the autocorrelation of temperature for a random node in the training data set. The lag granularity is one minute. The vertical dashed line indicates the mean application run duration, i.e., *2.3 hour*. The autocorrelation value of *lag=2.3h* for the temperature series is 0.5 while autocorrelation values are much stronger for smaller lags. This implies that the temperature series have strong temporal dependency.

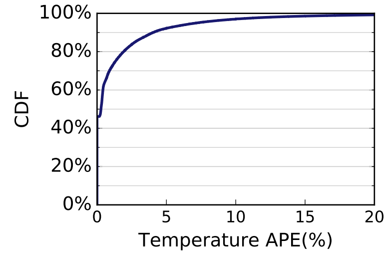
Figure 5.3(c) shows the comparison between real values and PRACTISE-predicted temperature series of the node shown in Figure 5.3(a). The temperature prediction is very close to the actual values. Yet, this is just the prediction across a short time window. Figure 5.3(b) illustrates the CDF of the absolute prediction error (APE) for the temperature data for the entire prediction week. APE is the absolute difference between actual value and prediction value divided by the actual value.

$$APE = \frac{|Actual - Prediction|}{Actual} \tag{5.5}$$

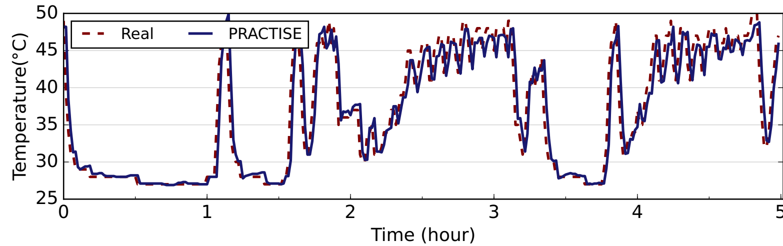
The smaller the APE, the better the accuracy of prediction. Figure 5.3(b) shows that for more than 90% of time, the APE is below 10%.



(a) Autocorrelation of temperature in the training set.



(b) PRACTISE prediction accuracy for temperature.



(c) PRACTISE temperature prediction overtime.

Figure 5.3: Autocorrelation and PRACTISE prediction for temperature.

5.4.3 SBE Prediction with PRACTISE

The above illustrates that PRACTISE can predict future temperature series accurately. As a next step, we apply the predicted temperature features to the neural network model, to test whether we can achieve good prediction of future SBE occurrences or not. All other features of the neural network model (node location, application) are known as well as duration and memory utilization (we use the average values from past runs of this application). Since we are interested in a fine granularity of prediction, i.e., on the specific node where the SBE may occur, we focus on a small set of cabinets. We choose 4 cabinets (384 nodes in total) in the upper left area (row 0 and 1 and column 6 and 7), which account for 10.4% of the total number of SBE occurrences in the entire sampling period.

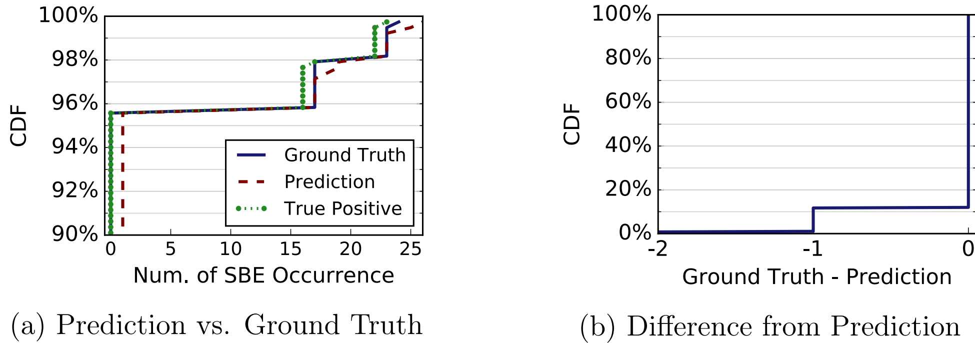
Table 5.2 shows the precision and recall for SBE occurrence prediction using real values (i.e., if we know the future temperature features) and PRACTISE-predicted temperatures. We observe that it is effective to leverage PRACTISE-predicted temperature values for prediction. The similar precision values indicate that using PRACTISE is able to achieve

the same level of correctness in prediction. While comparing recall values, the one with PRACTISE plugged-in is more conservative, reflected by the higher SBE recall and lower non-SBE recall.

Table 5.2: SBE Occurrence Prediction: Oracle vs. PRACTISE.

Input Type	Non-SBE		SBE	
	Precision	**Recall**	**Precision**	**Recall**
Oracle	0.86	0.72	0.82	0.92
PRACTISE	0.88	0.62	0.82	0.95

Similarly to Section 5.4.1 (see Figure 5.2), we compare the CDFs of SBE predictions per-node: ground truth, all predictions, and true positive predictions, see Figure 5.4(a). We can barely distinguish the three lines from one another, indicating that the prediction is remarkably accurate. Figure 5.4(b), shows the CDFs of the difference between ground truth and all predictions. For less than 20% of nodes, we over predict their SBE occurrences, but over-prediction is small (less than 2), especially comparing to the maximum number of SBE occurrences per node, which is around 25. 90% of predictions are exactly accurate.



(a) Prediction vs. Ground Truth (b) Difference from Prediction

Figure 5.4: Prediction for SBE occurrence at node level with PRACTISE.

In sum, we have shown that it is possible to accurately predict future SBE occurrences on specific nodes. This could have multiple applications including tuning the ECC turn on/off period on selected nodes and for selected applications, resulting in significantly reducing memory space and memory bandwidth overheads for many applications.

5.5 Discussion

In this section, we discuss the applications of the proposed SBE occurrence prediction tool. Meanwhile, we also demonstrate several open questions and challenges in this study and plans for future work.

5.5.1 Application of SBE Prediction.

An intuitive application of SBE prediction could be dynamically turning on/off the ECC mechanism on certain nodes for certain applications based on the prediction result. However, one may argue that it is too risky to completely turn off the ECC protection, especially for long-running scientific applications, as the aftermath of even a small probability of false positives is much more severe than the overhead of wastefully turning on ECC for a large portion of true negatives across the entire system.

Fortunately, there are several opportunities for bypassing this risk. First, we can leverage the fact that not all hardware errors will be reflected in the application outputs, which means that some of the errors are *masked*. Several show this by evaluating the impact of soft-errors, especially single bit errors, on GPU architecture with various fault-injection models and frameworks [154, 44, 59, 85]. For example, Hari et al. [59] build a compiler-based error injection, SASSIFI, and show that on average 80% of the injected single bit errors are actually masked in the output and thus are not perceived by the end user. Moreover, in [145], the authors claim that even for those corrupted outputs, there are chances that the outputs are acceptable by the end users. Though this work is done in the CPU domain, it is reasonable to assume that similar opportunities exist for GPU-accelerated applications. Note that, this idea of not-accurate but acceptable output is consistent with the goal pursued by scientists in approximate computing, including domains such as bioinformatics [96, 63], performance analysis [143], data mining [97], and image recognition [94]. Consequently, for those applications that do not require very strict accuracy, we can dynamically decide whether to turn on or turn off ECC protection based on our prediction

results. Clearly, we can always keep ECC on for those applications that need high-level of ECC protection. Therefore, by taking advantages of these opportunities, we are able to strike the balance between performance, overhead, and reliability.

5.5.2 Open Problems and Challenges

There are still several interesting open problems and challenges that are worthy of more detailed discussions. In this chapter, we perform feature selection based on the conclusions derived from the characterization section, as well as previous observations made by related works [140, 101, 55]. We demonstrate that the selected features all together are effective for SBE occurrence prediction. But it is also interesting to investigate which features or combinations of features play an irreplaceable role in prediction. Here, we leverage a neural network because of its powerful learning ability. Neural networks require a lot of computational capability and sometimes are prone to overfitting. Comparing our neural network solution with other machine learning models, such as SVM and decision trees, needs to be explored. Finally, SBEs show apparent spatial and temporal locality, which can also be leveraged by the prediction model.

5.6 Chapter Summary

In this chapter, we elaborate on how to exploit various related features for GPU SBE occurrence prediction. We propose a machine-learning-based technique that leverages observations of past system measurements to predict soft errors on Titan GPU nodes. We show that temperature and power consumption are of almost equal importance in GPU SBE occurrence prediction and together with a host of other factors including resource utilization, node location, and application type, are able to determine whether an upcoming application execution on a set of GPU nodes will result in SBEs or not. We evaluate our technique under various scenarios to demonstrate its effectiveness and robustness. In the next chapter, we explore the effectiveness of other machine learning models, such as

logistic regression, decision-tree-based models, and support vector machine, and compare their prediction quality.

Chapter 6

Predicting GPU Soft-Errors with a Variety of Machine Learning Models

In the previous chapter, we acknowledge the necessity of an error predictor for building a reliable system with lower error protection overhead and evaluate the effectiveness of a neural-network-based predictor. In this chapter, we explore the effectiveness of other machine learning models for predicting SBE occurrences on the Titan GPU nodes.

We first show how to select features in a systematical manner by categorizing them across space and time. Second, unlike the previous chapter, we overcome the imbalanced dataset challenge by taking advantage of inherent features of the dataset. Then, we use the selected features to train various machine learning models, including Logistic Regression (LR), Gradient Boosting Decision Tree (GBDT), Support Vector Machine (SVM), and Neural Network (NN).

Finally, we evaluate the machine learning models via different metrics and under diverse testing scenarios. Our results indicate that the proposed models achieve high prediction quality and are robust. In particular, the GBDT-based prediction achieves an F1 score of 0.81, significantly outperforming other models. The corresponding high recall (i.e., 0.87) and good precision (i.e., 0.76) indicate that the GBDT-based model is conservative in identifying as many SBE cases as possible. This is preferable as the aftermath of missing

an SBE occurrence is likely to be more severe than mislabeling a non-SBE occurrence. Our evaluation also uncovers interesting insights from comparison across different models, training/testing data, and feature combinations. We show that the proposed prediction models impose moderate overhead and are feasible in practice for GPU soft error prediction.

This chapter is organized as follows. In Section 6.1, we discuss how to select input features systematically from temporal and spatial perspectives. Section 6.2 illustrates the design details of building SBE occurrence predictors with a variety of machine learning models. We evaluate the proposed models under various conditions in Section 6.3. Section 6.4 summarized our findings.

6.1 Feature Selection

In this chapter, we focus on features that are related to GPU SBEs and organize them into time and space dimensions. The key premise is that soft errors are not an outcome that can be predicted by observing the instantaneous values of features. Therefore, it is important to include both temporal and spatial dimensions. Next, we list different features and their corresponding quantifiable representation.

6.1.1 Temporal Features

- **Application:** Some applications experience higher number of soft errors than others, indicating that application-specific features could be useful toward soft error prediction. We use application-specific features that can be obtained in non-intrusive manner, including the application binary name, total execution time (from past runs), and GPU resource utilization. GPU resource utilization includes the aggregate GPU core time, aggregate GPU memory, and maximum GPU memory. To capture the temporal behavior, we also use the application name that ran before the current execution to account for post-effects of an application run.

- **Temperature/power consumption:** We have shown evidence that temperature may be correlated with soft error occurrences in Chapter 4. However, capturing this complex correlation is non-trivial. We propose the following four temperature features to capture temporal aspects. First, we use the mean and standard deviation of the temperature during the current application run as two input features. In addition, to capture the dynamic behavior during a run, we use the mean and standard deviation of the *difference* between two consecutive temperature measurements as two additional input features. The above four features do not account for recent historical temperature behavior. To address this, we use temperature characteristics before the execution of a current application on the node. Specifically, we use the mean and standard deviation of the temperature series and the mean and standard deviation of the *difference* between two consecutive temperature measurements on the same node before the execution of the current application. We consider four time windows: 5min, 15min, 30min, and 60min prior to the start of the current execution to calculate the aforementioned four temperature features. Similarly, we apply the above described metrics for GPU power consumption.

6.1.2 Spatial Features

- **Node location:** Our characterization results indicate that error offender nodes are not uniformly distributed in space, and some error offenders experience SBEs repeatedly. Therefore, node location is used as a feature to capture node-specific and location-specific correlations.

- **Temperature/power consumption:** We have observed the prediction capabilities with temperature and power consumption on neighboring nodes. Similar to the representations of temperature and power consumption used in the temporal feature set, we leverage the mean and standard deviation of temperature and power consumption, as well as the mean and standard deviation of the difference between two

consecutive measurements for (1) the temperature of the CPU on the same node and (2) the temperature and power consumption of the GPU nodes in the same slot, as parts of the spatial feature set.

- **SBE history:** We include the error frequency in order to capture non-uniform temporal distribution of SBEs. Specifically, we use the total error count over the preceding day i.e., in the past 24 hours, at the node-level, and for the whole machine as features to capture the spatial behavior of error occurrence. We refer to this information as SBE rate history at the local (node) and global (whole machine) levels. We also include the SBE rate in the past 24 hours of the given application and the nodes allocated to it as additional history features.

6.2 Machine Learning Framework and Model

In this section, we focus on the discovery of the function that captures the relationship between input features and GPU soft error occurrences. To this end, we use several widely-used machine learning models including Logistic Regression (LR), Gradient Boosting Decision Tree (GBDT), Support Vector Machine (SVM), and Neural Network (NN). Our goal is to understand how the classification function can be learned effectively via carefully choosing a combination of features and an appropriate learning model, as well as what insights can be learned from evaluating such models.

6.2.1 Overview

The first step of the machine learning framework requires building the training dataset by collecting input features. In our case, we periodically collect information on input features for jobs running on the Titan. As a second step, this training dataset is used to build the machine learning model. The chosen model outputs the desired classification function that can be used for GPU soft error prediction. The desired classification function is a two-class classifier (i.e., whether an error occurs or not during the target program execution), and is

dependent on the training dataset and the selected model. Building the training dataset and estimating the classification function is an iterative process that aims to refine the learned classification function as time passes. Here, the model construction is relatively less frequent (i.e., once every two weeks). The final step is to feed the features of the target program into the models to predict error occurrence.

Here, we encounter the same problem as in Chapter 5: some input features for the target application run can be collected prior to execution (e.g., machine-level error rate, node specific characteristics) while certain program specific features such as GPU power and temperature profiles can not always be known a priori. Similarly, we experiment with two approaches (see also Section 5.4 for additional detailed discussion):

- Predict *at the end of* the application execution so that we know all input features correctly.
- Predict *before* the application execution and infer those input features that cannot be known a priori with time-series prediction tools, e.g., [28, 151].

In Chapter 5.4, we have shown that the temperature and power consumption profile can be accurately captured by PRACTISE [151], which is a neural-network-based time series prediction tool. Moreover, we observe that the model trained with the values predicted by PRACTISE is able to achieve very close prediction quality as using oracle values.

6.2.2 Challenge: Imbalanced Dataset

It is desired to select the training and testing data so that they cover a wide variety of workload and system properties, and are also representative of a real-world scenario. In our approach, any workload execution that uses GPU resources is a qualified sample. This ensures that our dataset corresponds to different kinds of workloads distributed over both time and space dimensions. However, as stated in Section 5.3.2, this data collection approach results in a challenging problem: a highly imbalanced dataset. The problem stems for the fact that only a limited number ($\leq 2\%$ in our case) of application runs

encounter SBEs. This makes the size of majority class (SBE-free samples) much larger than that of the minority class, which is our focus.

Mitigating the imbalanced dataset challenge usually has two solutions. The first one is over-sampling the minority class, i.e., by generating synthetic samples [120, 30]. The other solution is to under-sample the majority class, i.e., by randomly choosing a subset of samples [132] or control under-sampling via clustering algorithms such as k-means [27]. In Chapter 5, we propose a customized under-sampling algorithm that is able to effectively reduce the size of the majority class (see Section 5.3.2). However, none of these methods take the inherent dataset features into consideration. In the following section, we propose a two-stage method, which first leverages the dataset characteristics to mitigate the challenge of the imbalanced dataset and then apply machine learning models to predict SBE occurrences.

6.2.3 Two-Stage Machine Learning Models

In Chapter 4, we observe that a small fraction of GPU nodes and workloads are responsible for a large number of SBEs. It is intuitive to think that previous SBE-affected nodes/-workloads may continue seeing SBEs while those SBE-free nodes/workloads are likely to remain in "safe status" in the future. Accordingly, we consider three basic schemes: *Basic A* predicts that any application run involving a SBE offender node will result in a SBE-affected run. *Basic B* predicts that previously SBE-affected applications will result in future SBE-affected runs. *Basic C* predicts that top SBE-affected applications will result in a future SBE-affected run. Top SBE affected applications are defined as the top 20% applications that encounter SBEs in the training phase in terms of their total number of SBEs.

Table 6.1 reports the precision and recall scores for the above three basic schemes, compared to a trivial random classifier that assumes the probability of encountering SBEs is 0.5. The random classifier achieves a mere 0.5 recall. Due to the high imbalance between the two classes, the random classifier achieves very low precision for the SBE class

prediction. *Basic A* significantly outperforms the random classifier and the other two basic schemes, achieving a high SBE prediction recall (0.94), albeit at fairly low precision (0.40). This indicates that the scheme *Basic A* could capture most SBE cases but still over-predicts the SBE class, implying that this scheme alone is insufficient for robust prediction.

Table 6.1: Precision and recall for basic schemes.

Scheme	**SBE Sample**		**Non-SBE Sample**	
	Precision	**Recall**	**Precision**	**Recall**
Random	0.02	0.50	0.98	0.50
Basic A	0.40	0.94	0.99	0.98
Basic B	0.02	0.69	0.98	0.24
Basic C	0.00	0.06	0.98	0.76

Inspired by *Basic A*, which achieves a reasonable prediction quality, we derive a *TwoStage* method. This method leverages the inherent temporal dependency of our dataset and takes advantage of the power of machine learning techniques. Unlike *Basic A*, the *TwoStage* method is able to accurately predict the samples from SBE offenders, instead of blindly assuming them to always encounter SBEs in the future. During training, we train the model solely on samples from SBE offender nodes. The prediction flow is presented in Fig. 6.1.

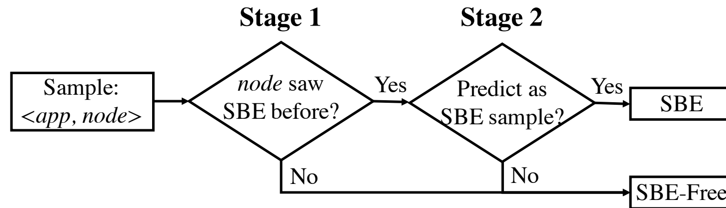


Figure 6.1: *TwoStage* method: prediction flow.

At the first stage, samples are checked to see if they come from SBE or non-SBE offender nodes. They are passed to the second stage only if they come from SBE offender nodes. The advantages of this method are three-fold: (1) the number of SBE offender nodes is much smaller than the number of non-SBE offender nodes. Therefore, this step

automatically reduces the training data size, resulting in less training overhead (both in terms of time and storage). (2) As discussed previously, the relationship between SBEs and different features is complex. By focusing on SBE offender nodes only, we avoid the noise and interference from error-free samples. (3) Most importantly, this approach solves the problem of data imbalance. Now, after the first stage, the ratio between SBE-free samples and SBE-affected ones is roughly $2:1$ (consider that the original ratio is almost $50:1$). The downside is that this method always misses SBE occurrences on previously error-free nodes. Fortunately, on the Titan, such probability is low and frequent periodic training of the model resolves this issue. Section 5.4 shows that *TwoStage* introduces low overhead and can be trained periodically to provide high prediction quality.

6.2.4 Machine Learning Model Selection

We select four widely used machine learning models that provide a wide variety of trade-offs and advantages. **Logistic Regression (LR)** is a simple and fast model for understanding the influence of several independent variables but limited by the linear function between inputs and outputs. **Gradient Boosting Decision Tree (GBDT)** is a boosting-based model that is essentially an ensemble of weak models. GBDT is effective in tackling the variance-bias problem, but is computationally expensive. **Support Vector Machine (SVM)** is designed to solve classification problems by performing non-linear classification using a kernel. **Artificial Neural Networks (NN)** are inspired by biological neural networks and are composed of many interconnected neurons. The weights associated with the neurons are used to approximate non-linear functions of the input. Neural networks capture the complex pattern between features and targets. In the evaluation section (Section 6.3), we incorporate the aforementioned models to the *TwoStage* method and compare their effectiveness.

6.3 Evaluation and Analysis

Before discussing the prediction results, we describe the data used for model training and testing, as well as the evaluation metrics.

6.3.1 Data Description and Evaluation Metrics

In this chapter, we use the same system data as in Chapter 5. We collect all features discussed in Section 6.1 over the entire sampling period (from January to June, 2015) for both SBE-affected and SBE-free periods. Unlike Chapter 5, we divide this dataset into three pairs of training and testing sub-datasets based on the time dimension so that we can also evaluate the robustness of *TwoStage*. In each sub-dataset, the training dataset consists of 3.5-month samples. The samples in the following two weeks are used for testing. Each sample is identified as the pair of the application name and the node ID. For example, our first training dataset (i.e., DS1) corresponds to 6.7 thousand application executions, with roughly 5 million samples. Note that each application run may produce multiple number of samples depending on the number of nodes allocated during the execution. For determining the length of the training and testing datasets, we follow the rule-of-thumb ratio of the testing data size to the training data size ($20\% - 25\%$) [56]. We also ensure that the three testing datasets cover diverse workloads and have different compositions of samples.

In order to meaningfully evaluate the results, we use all three metrics described in Section 5.2: precision, recall, and F1 score. We focus on F1 score, the harmonic mean of precision and recall, to capture the trade-off between prediction and recall and evaluate prediction quality across different models. F1 score allows for easy comparisons across models using a single metric.

6.3.2 Machine Learning Model Comparison

As stated in Section 6.2.4, we apply four machine learning models (i.e., LR, GBDT, SVM, and NN) on the second stage of the *TwoStage* method. Here, we discuss which machine learning model works most efficiently.

Accuracy and robustness comparison: Choosing an effective model is one of the key challenges. Fig. 6.2 reveals the F1 score of SBE class using the first dataset (DS1) for the four machine learning models. Note that the result of SBE-free class is not shown (also in later evaluation parts) because all models are able to achieve high prediction quality for the SBE-free cases (i.e., the majority class) due to the highly imbalanced nature of our testing samples.

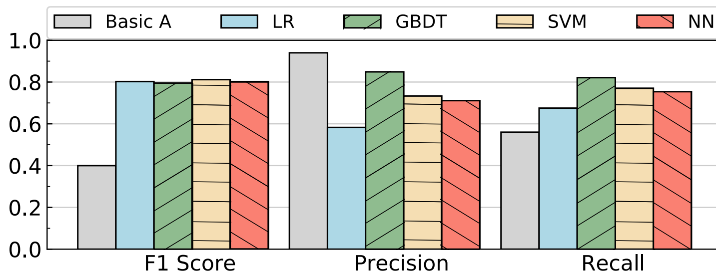


Figure 6.2: Comparison of SBE occurrence prediction across different models for DS1.

We notice that applying machine learning models always significantly surpasses the *Basic A* scheme, with at least 0.1 improvement for the F1 score. Applying GBDT achieves the highest F1 score (0.81), outperforming the least effective one (LR) by 0.14. To investigate why GBDT works better than the other models, we also look at the precision and recall values. We find that all four models are able to achieve similar precision values (around 0.8), but GBDT is able to achieve a much higher recall value (0.87) than the other three models (around 0.6). A high recall value implies that the boosting nature of GBDT enables it to identify more SBE samples, while similar precision across four different learning models indicates that GBDT also conservatively predicts SBE occurrences as the other three models. This result suggests that GBDT achieves the most accurate

prediction of SBE occurrences among the four machine learning models.

Across different datasets: We have shown that applying GBDT yields to the best prediction result for the first dataset. Here we validate whether GBDT works best for other datasets (i.e., DS2 and DS3). Note that these testing and training datasets are disjoint and the machine learning models are trained independently for each dataset. Table 6.2 summarizes the F1 scores for the other two datasets. First, the table shows that applying machine learning models almost always leads to improvement in the F1 score, compared with *Basic A*. Secondly, using GBDT results in satisfactory prediction quality (F1 score) across different datasets and significantly outperforms all the other three models. Even for the most tough-to-predict dataset (DS3), applying GBDT within *TwoStage* improves the F1 score to 0.71. The above observations confirm the efficiency and robustness of GBDT.

Table 6.2: F1 score for SBE occurrence prediction.

Dataset	Basic A	LR	GBDT	SVM	NN
DS1	0.56	0.67	**0.81**	0.70	0.69
DS2	0.75	0.80	**0.81**	0.79	0.77
DS3	0.55	0.52	**0.71**	0.55	0.51

Model overhead comparison: In the previous subsection, we have illustrated that the *TwoStage* method with GBDT is effective and robust. Here, we evaluate its training overhead, especially since the Titan operation would require re-training to occur periodically. The comparison of the training time of the four machine learning models is presented in Table 6.3. Note that all experiments are conducted on an Intel Xeon server (Intel E5-4627v2) with 512GB RAM. The training time is the longest for SVM and is approximately one hour. This is due to the computationally expensive quadratic RBF kernel used in the SVM model. LR consumes the least amount of time, but it also fails to provide a guaranteed prediction quality (see Fig. 6.2 and Table 6.2). Considering both prediction quality and overhead, GBDT is superior as it strikes a good balance between these two measures. Note that since the training process can be done offline and periodically (e.g.,

repeated every two weeks), the relative model re-training time is truly negligible. Overall, GBDT's small training time would allow re-training to happen even several times during the day if needed. In addition, the data movement overhead for storing and preprocessing the data is of the order of minutes.

Table 6.3: Mean training time for various models.

	LR	**GBDT**	**SVM**	**NN**
Model Mean Time	4.81 s	40.53 s	1.04 h	20.01 min

The aforementioned evidence supports that *TwoStage* with GBDT is practically feasible for error prediction. In the later sections, we show prediction results based on this model only.

6.3.3 Feature Analysis

Besides choosing an appropriate machine learning model, the selection of features is another key to achieving high-quality of prediction. In Section 6.1, we illustrate several features from temporal and spatial perspectives, which may contribute to the SBE occurrence prediction. This does not imply that all features are needed for training the most effective model. Nonetheless, it is non-trivial to discover and engineer the feature set resulting in the highest prediction quality. In this section, we explain how to perform the feature discovery process.

The large number of features and complexity of advanced learning models make it challenging to meaningfully understand the impact of each feature. Consequently, we simplify this problem by grouping features into categories (feature groups) and train the machine learning models with each feature group. The goal is to see which feature group contributes most to the prediction quality. We also train one model with *all* features. Fig. 6.3 shows the effect of different feature groups on the prediction quality, in the form of the percentage improvement for the F1 score comparing to *Basic A*. The labels in the figure legend indicate the corresponding feature groups used in each experiment.

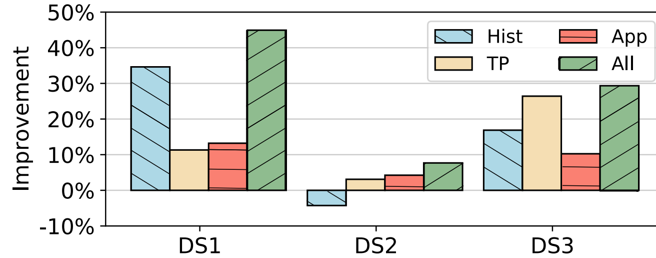


Figure 6.3: Effect of different feature groups on F1 score, in terms of the improvement over *Basic A*. **All** means using all features discussed in Section 6.1. **Hist**, **TP**, and **App** correspond to SBE history, temperature/power consumption, and application-related features, respectively.

We observe that almost all models trained with any feature group positively contribute to the SBE occurrence prediction, but with different degrees of improvement. Meanwhile, no single feature group is the winner across all datasets. For example, Hist is the most effective feature group for DS1, but it negatively impacts prediction quality in DS2. However, in all datasets, using the combination of all features always results in the biggest improvement, implying that all features are valuable and needed for achieving good prediction.

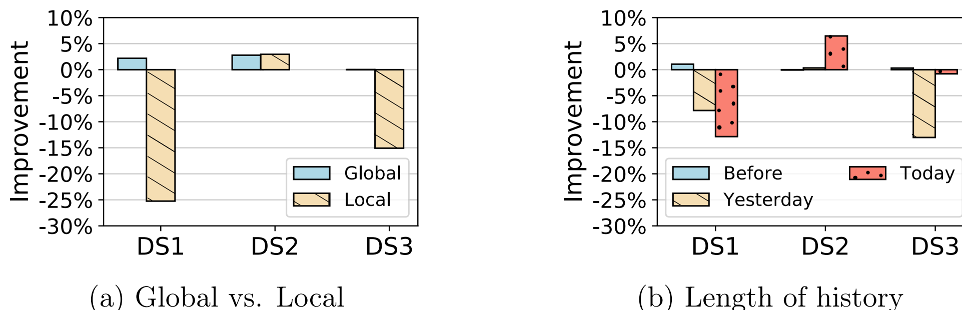
Besides feature grouping, it is also interesting to conduct a deeper and more fine-grained investigation on input features. We start by quantifying the impact of various types of temperature/power consumption features. As stated in Section 6.1, temperature and power consumption features are collected from both temporal and spatial perspectives, on the targeted node and other neighboring nodes in the same slot. Therefore, we conduct experiments with various combination of temperature and power consumption features to see their impact on SBE occurrence prediction, see Table 6.4. Cur refers to using temperature and power consumption data collected only from the targeted node during the application run, together with all other groups of features mentioned in Section 6.1. In addition to the features used in Cur, CurPrev also leverages temperature and power consumption data prior to the execution of application on the targeted node (in four time windows, up to one hour). Similarly, CurNei adds the temperature and power consumption data on neighboring nodes (i.e., in the same slot as the targeted node). CurPrevNei leverages

all temperature and power consumption features discussed above. Interestingly, we notice that the prediction quality is not significantly affected by the various feature combinations. Looking at F1 score, CurPrev and CurPrevNei work worse than Cur. In contrast, CurNei achieves slightly better prediction quality, but it also leverages more features which means it introduces more overhead in terms of data collection and model training. Cur exhibits high recall and good precision. Consequently, we select Cur as an effective and light-weight representation of temperature and power consumption information for model training.

Table 6.4: Effect from temporal and spatial aspects of temperature and power features.

Feature Set	Precision	Recall	F1 Score
Cur	0.764	0.865	0.820
CurPrev	0.801	0.830	0.815
CurNei	0.815	0.838	0.826
CurPrevNei	0.807	0.829	0.818

As a next step, we analyze the impact of various types of history features on the SBE occurrence prediction. Unlike the aforementioned experiments, here we conduct the experiment by *removing* one type of history features and see the decrease in F1 score. First, we compare the effects from global (overall information collected from the whole system) and local (information collected from the targeted node) SBE history on SBE occurrence prediction, see Fig. 6.4(a). Interestingly, removing global and local history even increases the F1 score in DS2, which is consistent with the observation in Fig. 6.3, where SBE history features contribute negatively in DS2. However, if we focus on DS1 and DS3, we notice that local history information plays a more important role in prediction, i.e., removing these features leads to 15% to 25% loss in F1 score. The impact of history length on prediction quality is presented in Fig. 6.4 (b). From this figure, we observe that the importance of SBE history generally increases as it is closer to the current time. Note also that there is no particular length (i.e., today, yesterday, or full history) that is always effective across all datasets. This illustrates the importance of inclusion of all SBE history features.



(a) Global vs. Local

(b) Length of history

Figure 6.4: Decrement on F1 score if removing a certain feature set from the original feature combination: global vs local (a), and different length of SBE history (b).

6.3.4 Prediction Analysis

In the previous sections, we have determined that GBDT is the best machine learning model for the *TwoStage* method, and the most effective feature combination for its training. Here, we conduct an evaluation on the prediction quality of this model with the most efficient feature combination as inputs. We illustrate the analysis on the results of using the first dataset only. The quality of prediction for the two other datasets is similar to that of DS1.

Spatial robustness: We investigate if *TwoStage* performs well spatially across the entire Titan system. Fig. 6.5(a) shows the proximity of the cumulative distribution plots of SBE predictions across the entire system for the ground truth, prediction (true positives plus false positives) and true positives. We then present the absolute difference between the number of SBE affected application runs (ground truth) and the prediction for the testing period at the cabinet level, see Fig. 6.5(b). For over 95% of cabinets, the error difference is relatively small, ranging in $[-15, 13]$. In fact, there are only 3 (out of 200) cabinets where the prediction overestimates SBE affected application runs by more than 25. This is encouraging as thousands of applications are executed over each cabinet. We also perform such analysis at the node level and observe accurate prediction for more than 99% of nodes.

We also investigate how the choice of optimal model changes across the various cabinets.

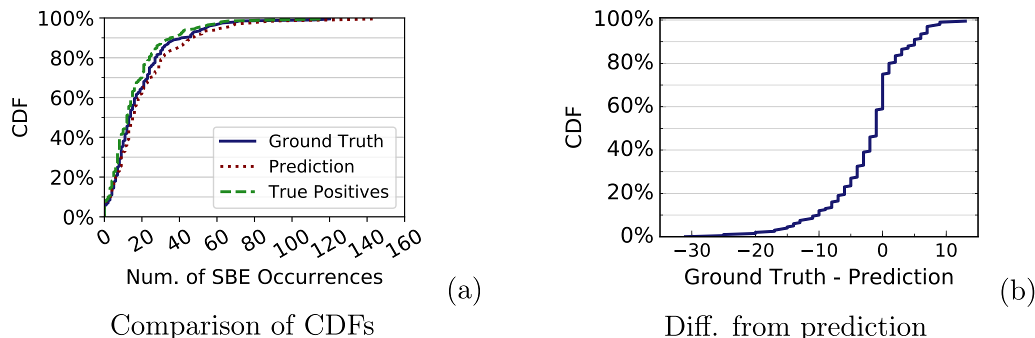


Figure 6.5: Comparison between SBE occurrence prediction and ground truth at the cabinet level.

We find that *TwoStage* with GBDT remains the close-to-the-best choice among all models for all cabinets. The number of cabinets where this scheme is not the optimal choice is limited across the machine in all three datasets. In fact, we find that even if the prediction model is chosen with the apriori knowledge (oracle) on the optimal model, the overall F1 score improves only by 0.01, 0.02, and 0.001 for the three datasets, respectively. Overall, our results indicate that *TwoStage* with GBDT delivers robust and consistent results across the whole machine and it is not restricted to performing well only in selected sections of the machine.

Effect of application runtime: We look into whether the quality of the prediction is significantly impacted by the length of the application execution. In other words, do short-running and long-running applications attain comparable prediction quality? We classify an application as "short-running" if its runtime falls in the bottom 25 percentile range and as "long-running" if its runtime falls in the top 25 percentile range. Table 6.5 confirms that both types of application achieve high prediction quality with comparable F1 scores. Moreover, "long-running" applications achieve better prediction quality than "short-running" ones. This is quite favorable since the cost of mislabeling a "long-running" application would be higher, e.g., if re-execution is needed.

Effect of SBE severity:

An error predictor that is able to label more severe application runs (i.e., with a higher

Table 6.5: SBE occurrence prediction for "short-running" and "long-running" applications.

Application	Precision	Recall	F1 Score
All	0.76	0.87	0.81
Short	0.77	0.94	0.84
Long	0.93	0.90	0.92

number of SBEs) as SBE-affected is desirable. Towards this goal, we first group application runs into four levels of SBE severity (25 percentile per level), i.e., the bottom 25 percentile applications with the least number of SBEs are in *Light* while the top 25 percentile ones are in level *Extreme*. Table 6.6 presents the percentage of correctly classified SBE-affected runs in each level. Our results indicate that as the number of SBEs increases among application runs in our dataset, the effectiveness of the *TwoStage* method grows. For example, 74% of the application runs in level *Light* are already correctly predicted to be SBE-affected cases. The percentage number increases as the SBE severity level goes higher, becoming 95% for *Extreme* application runs. The results show that *TwoStage* is able to achieve high prediction quality for SBE occurrences, especially for those applications affected by more SBEs.

Table 6.6: Percentage of correctly classified SBE-affected application runs in four severity levels.

	Light (<25%-ile)	**Moderate (25-50%-ile)**	**Severe (50-75%-ile)**	**Extreme (>75%-ile)**
Severity PCT.	74%	88%	93%	95%

6.4 Chapter Summary

In this chapter, we propose several machine learning-based models that use workload and system features as input for GPU soft-error prediction. We overcome the imbalanced dataset challenge by taking advantage of the inherent feature present in our dataset. We examine their effectiveness under various scenarios and in multiple aspects including

accuracy, robustness, overhead, and model interpretations.

Chapter 7

Fault Site Pruning for Practical Reliability Analysis of GPGPU Applications

In the previous chapters, we take a large-scale study on the GPU soft errors on America's fastest supercomputer – Titan, with a special emphasize on the single-bit errors. Through the characterizing analysis, we observe that GPU soft errors are related to many features, including workloads, resource utilization, temperature, and power consumption (see Chapters 3 and 4). In Chapters 5 and 6, we propose two SBE occurrence predictors on top of four commonly-used machine learning models. These two chapters address the problem from the perspective of the entire system. Still, such analysis has its inherent drawback. The large-scale system measurement data are *post hoc*, i.e., we have limited control over the data collection methodology and we cannot use our techniques to dynamically turn on/off ECC. In this chapter, we adopt an application-level view of the GPU reliability problem. In Chapters 3 and 4, we observe that different applications experience different rates of bit flips, possibly due to their data access pattern and interaction with hardware. In this chapter, we aim to deepen this understanding. Here, we consider analysis at the application-level to explore the reasons why some applications are more resilient

to GPU soft errors than others. Such understanding is helpful in building reliable GPU architectures.

Past work [154, 44, 59] investigates the error resilience of GPU applications. A popular way is to understand resilience by artificially but systematically injecting faults into various register states or logic units and then by examining their effects on the application output. These faults can result in: a) no change in application output (i.e., faults are masked), b) change in application's output due to data corruption but still execution terminates successfully (i.e., faults are silent), and c) application crashes and hangs. The latter two outcomes are certainly not desirable from the reliability point-of-view and hence a lot of high-overhead protection mechanisms such as check-pointing [137, 80] and error correction codes (ECC) [8, 10, 4] are employed to strive for reliable executions.

One of the major challenges in evaluating error resilience of applications is to obtain a very high fault coverage, i.e., inject faults in all possible fault sites and record its effect. This procedure is already very time consuming and tedious. In our own analysis of GPGPU applications, we have found that the total number of fault sites can be in the *order of billions*. Assuming a single-bit flip model, Table 7.1 quantifies the total number of fault injection sites for a large number of diverse GPGPU application kernels. The tremendous size of fault sites is due to the fact that each GPGPU kernel can spawn thousands of application threads and each thread is assigned to a dedicated amount of on-chip resources. For the calculation of fault sites reported in Table 7.1, we only consider soft errors that can occur in functional units (e.g., arithmetic logic unit and load-store unit) [59]. Yet, the number of fault sites is tremendous. Executing one experiment per fault site in such a vast space to collect application error resilience metrics is clearly very difficult and absolutely not practical.

In order to develop a robust and practical reliability evaluation for GPUs, prior works have considered a variety of fault injection methodologies such as LLFI-GPU [85] and SASSIFI [59] that sample a subset of fault sites to capture a partial view of the overall error resilience characteristics of GPGPU applications. These works claim that experiments

Table 7.1: Various metrics (including the total number of possible fault sites) related to considered GPGPU application kernels.

Suite	Application	Kernel Name	ID	# Threads	# Total Fault Sites
Rodinia	HotSpot	calculate_temp	K1	9216	3.44E+07
	K-Means	invert_mapping	K1	2304	1.47E+07
		kmeansPoint	K2	2304	9.67E+07
	Gaussian Elimination	Fan1	K1	512	1.63E+05
		Fan2	K2	4096	4.92E+06
		Fan1	K125	512	1.09E+05
		Fan2	K126	4096	8.79E+05
	PathFinder	dynproc_kernel	K1	1280	2.77E+07
	LU Decomposition (LUD)	lud_perimeter	K44	32	1.75E+06
		lud_internal	K45	256	6.84E+05
		lud_diagonal	K46	16	5.26E+05
Polybench	2DCONV	Convolution2D_kernel	K1	8192	6.32E+06
	MVT	mvt_kernel1	K1	512	6.83E+07
	2MM	mm2_kernel1	K1	16384	5.55E+08
	GEMM	gemm_kernel	K1	16384	6.23E+08
	SYRK	syrk_kernel	K1	16384	6.23E+08

on a small and randomly selected set of fault sites is sufficient for results within 95% confidence intervals and error margins within a 6% range [83]. In this chapter, we take an orthogonal approach – our goal is to prune the large amount of fault site space via carefully considering the properties of GPGPU applications. Our pruning mechanisms not only reduce the total number of required fault injections (in some cases to a few hundreds only while still maintaining superior accuracy), but also equivalently reduce the total time to complete the required experiments.

To this end, we focus on the following fundamental observations relevant to GPGPU applications: a) GPGPU applications follow the SIMT execution style that allow many threads to execute the same set of instructions with slightly different input values, b) There is an ample commonality in code across different threads, c) Each GPU thread can have several loop iterations that do not necessarily change the register states significantly, and d)

GPGPU applications themselves are error resilient and hence changes in the precision/accuracy of register values do not necessarily change the final output of an application. By leveraging these properties, we propose *progressive* pruning that systematically reduces the number of fault sites while preserving the application error resilience characteristics. Our proposed methodology consists of:

- *Thread-wise Pruning:* The first step focuses on reducing the number of threads for fault injection. We find that a lot of threads in a kernel have similar error resilience characteristics because they execute the same number and type of dynamic instructions. Based on the grouping of threads based on dynamic instruction count, we select a small set of representative threads per kernel and prune the redundant fault sites belonging to other threads.

- *Instruction-wise Pruning:* Our detailed analysis show that many of these selected representative threads still execute subsets of dynamic instructions that are identical across threads. This implies that all instructions are not required to be considered for fault injection, and that the replicated subsets across threads can be considered only once. Therefore, the replicated fault sites are further pruned while preserving the application error resilience characteristics.

- *Loop-wise and Bit-wise Pruning:* We observe that there is a significant redundancy in fault sites across loop iterations and register bit positions. Therefore, such redundant fault sites can be further pruned for further savings while accurately capturing the application error resilience characteristics.

To the best of our knowledge, this is the first work that quantifies the problem of high number of fault sites in GPUs and develops progressive pruning techniques by leveraging GPGPU application-specific properties. Our newly proposed methodology is able to reduce the fault site space by up to seven orders of magnitude while maintaining accuracy that is close to that of ground truth. We further extend the progressive fault site pruning technique to multi-bit fault model and investigate the impact of increasing number of faults on application outcomes.

This chapter is organized as follows. In Section 7.1, we discuss related work. Section 7.2 explains the background and methodology. We illustrate the design details of the progressive fault site pruning in Section 7.3 and evaluate it in Section 7.4. In Section 7.5, we extend the pruning technique to the multi-bit fault model and evaluate the impact of multiple faults on the outcomes of GPGPU applications. Finally, Section 7.6 summarizes the chapter.

7.1 Related Work

To the best of our knowledge, this is the first work that identifies the problem of large number of fault sites that make GPU reliability assessment impractical and proposes ways to efficiently address it. In this section, we briefly discuss works that are most relevant to this study.

High-level Reliability Analysis. Simulation-based analysis is employed widely in characterizing critical hardware structures for the purpose of finding vulnerabilities introduced by soft errors. Prior work [45, 64, 138] performed architectural vulnerability analysis (AVF) by performing exhaustive fault injection experiments. For the analysis purposes, faults are injected at various levels (e.g., application- or micro-architecture-level) and the effects of bit flips are measured by analyzing the application output. Application-level fault injection techniques are widely used in evaluating error resilience characteristics for both CPU [32, 153] and GPU applications [154]. They are generally fast and still can provide detailed information. However, Cho et al. [33] pointed out that application-level methods can be inaccurate as compared to flip-flop-level methods for CPU applications. Another option is performing neutron-beam experiments [47], which is not always feasible. We acknowledge the aforementioned pros and cons of various techniques for reliability analysis. In this chapter, we follow the process of studying reliability via fault injection, at PTXPlus-level, which is much faster and feasible than beam injection and is also reasonably accurate [144]. The aforementioned studies adopt the commonly used single-bit fault

model. Still, it is worthwhile to investigate multi-bit fault model. Sangchoolie et al. [124] look into the impact of multi-bit faults for CPU applications. In this chapter, we start with single-bit fault model and extend the proposed methodology to multi-bit fault model for GPU applications.

Fault Injection Analysis. Although much work has been done on fault injector models/frameworks [92, 42, 88, 89, 113, 117, 123, 128, 126, 23, 95, 49, 50, 51] in the CPU domain, there are only a limited number of fault injection models designed specifically for GPUs. For example, to evaluate application error resilience in GPUs, Fang et al. [44] proposed GPU-Qin to understand how faults affect an application's output in GPUs. A GPU debugging tool *cuda-gdb* [3] is leveraged by GPU-Qin to inject single bit errors into the destination operands. Similarly, Hari et al. [59] developed a fault injection tool, called SASSIFI, which injects different kinds of faults into destination register values, destination register indices and store addresses, and register files.

Fault-site Pruning. One of the major concerns of aforementioned fault injection works, both in CPU and GPU domain, is the space complexity of possible fault sites. Within the CPU context, major works by Relyzer [58] and MeRLiN [70] grouped fault sites into equivalence classes and select one or more pilots per class for fault injection. They showed significant benefits of employing their mechanisms in the workloads typically executed on CPUs. We believe directly transferring such pruning techniques to GPU applications is not straightforward because GPU applications typically spawn hundreds to thousands of threads, leading to enormous fault site space. Our work identifies fruitful features that play a role in the final error resilience characteristics of an application and leverage them to carefully prune the fault site space. Finally, to illustrate the effectiveness of our pruning mechanisms, we performed exhaustive experiments on the pruned space and compared the results to the ones closest to the ground truth.

7.2 Background and Methodology

We selected applications from commonly used suites (i.e., Rodinia [31] and Polybench [54]) that cover a variety of workloads from different domains. Note that, as kernels of GPGPU applications normally implement independent modules/functions, we perform resilience analysis separately for each kernel. We focus on every static kernel in the application. For static kernels with more than one dynamic invocations, we randomly select one for fault injection experiments. Table 7.1 shows the evaluated 10 applications (16 kernels). In the rest of this chapter, if the kernel index is not specified, it implies that the application contains only one kernel.

7.2.1 Baseline Fault Injection Methodology

We employed a robust fault injection methodology based on GPGPU-Sim [24], a widely-used cycle-level GPU architectural simulator. The usability of GPGPU-Sim with PTXPlus mode (which provides a one-to-one instruction mapping to actual ISA for GPUs [144, 24]) for reliability assessment is validated by GUFI [144], a GPGPU-Sim based framework. In this work, we inject faults using GPGPU-Sim with the PTXPlus mode.

For each experiment, we examine the application output to understand the effect of an injected fault. We classify the outcome of a fault injection into one of the three categories: (1) *masked output*, where the injected fault leads to no change in the application output, (2) *silent data corruption (SDC) output*, where the injected fault allows the application to complete successfully albeit with an incorrect output, and (3) *other output*, where the injected fault results in application hangs or crashes. The distribution (or percentage) of fault injection outcomes in these three different categories form the error resilience profile (or characteristics) of a GPGPU application.

7.2.2 Baseline Fault Model

We focus on injecting faults in the destination registers to mimic the effect of soft errors occurred in the functional units (e.g., arithmetic and logic units (ALUs) and the load-store units (LSUs)) [44, 59]. The destination registers and associated storage are identified by thread id, instruction id, and bit position. Table 7.1 shows a few characteristics of various application kernels, including the number of threads spawned by each kernel and the total number of fault sites (also called fault coverage). The fault coverage for each application kernel (consisting of N threads) is calculated as per Equation (7.1). Suppose that a target thread t ($t \in [1, N]$) consists of $M(t)$ dynamic instructions and that the number of bits in the destination register of instruction i ($i \in [1, M(t)]$) is $bit(t, i)$. The number of exhaustive fault sites is the summation of every bit in every instruction from every thread in the kernel and is given by:

$$FaultCoverage = \sum_{t=1}^{N} \sum_{i=1}^{M(t)} bit(t, i). \tag{7.1}$$

This number for the GPGPU kernels that we consider in this chapter is reported in the rightmost column of Table 7.1. Recall that, the numbers are obtained under the context of single-bit fault model, which is the commonly used model in most fault injection studies [44, 59]. In this chapter, we study both single-bit and multi-bit fault models. We start with single-bit fault model to build a fault site pruning technique. Then, we explain how to extend the technique to multi-bit fault model.

7.2.3 Statistical Considerations

Looking at the number of exhaustive fault sites shown in Table 7.1, it is clear that it is not practical to perform fault injection runs for all fault sites. This is especially true when application execution time is very long, which is especially true for production software or workloads executing in data centers [121]). Taking GEMM from Polybench as an example and assuming that it takes a (nominal) one minute to execute one fault injection experiment, then 7.73E+08 minutes (or about 1331 years) are needed to complete experiments

for the entire fault site space (see the first row in Table 7.2). Therefore, it is desirable to reduce the number of fault injection experiments but also guarantee a statistically sound resilience profile (i.e., percentages of masked, SDC, and other outputs – see Section 7.2.1) of the considered application kernel. To this end, prior work [83] has shown that given an initial population size N (in our case, N is the number of exhaustive fault sites), a desired error margin e, and a confidence interval (expressed by the t-statistic), the number of required experiments n (in our case, fault sites) is given by:

$$n = \frac{N}{1 + e^2 \times \frac{N-1}{t^2 \times p \times (1-p)}} \tag{7.2}$$

Note that p in the above equation is the program vulnerability factor, i.e., the percentage of fault injection outcomes that are in the masked output category. If $n \ll N$, (e.g., if the percentage of samples is less than 5% of the entire population), then N can be approximated by ∞, resulting in the following equation [82]:

$$\lim_{N \to \infty} n = \lim_{N \to \infty} \frac{N}{1 + e^2 \times \frac{N-1}{t^2 \times p \times (1-p)}} = \frac{t^2}{e^2} \times p \times (1-p). \tag{7.3}$$

Since p is the result of fault injection experiments, p is still unknown. To ensure that the number of fault injection experiments n is sufficient to capture the true p [83], then

$$n = max\{\frac{t^2}{e^2} \times p \times (1-p)\} = \frac{t^2}{4 \times e^2}, \tag{7.4}$$

where n is the minimum sample size (i.e., number of fault injection experiments) required to calculate the fraction of fault injection outcomes in the masked output category, with a certain confidence interval and a user-given error margin e. To maximize the term $p \times (1-p)$, p is set to 0.5.

Table 7.2 presents the required number of fault injection experiments (i.e., fault sites) in GEMM given a confidence interval and an error margin. We consider the reliability profile results of 60K experiments (with 99.8% confidence interval and an error margin of

Table 7.2: Fault sites and other statistics for GEMM.

Confidence Interval	Error Margin	# Fault Sites	Estimated Time	Masked Output (%)
100%	0.0%	7.73E+08	1331 years	?
99.8%	±0.63%	60,181	40 days	24.2%
95%	±3.0%	1,062	16 hours	21.6%

$e = 0.63\%$) as the *ground truth* [70]. Clearly, there is a significant discrepancy between the percentage of *masked* outputs for 60K versus 1K fault injections (see last column). The goal of our fault site pruning mechanism is to achieve the accuracy of the 60K results but with a much reduced number of experiments.

7.3 Progressive Fault Site Pruning

In this section, we explain the proposed error site pruning techniques while providing intuition along the steps.

7.3.1 Overview

Figure 7.1 provides an overview of our fault site pruning four-stage mechanism. This mechanism is progressive, i.e., every successive stage further reduces the number of fault sites of the previous one. There are four primary stages: *a) Thread-wise Pruning*, *b) Instruction-wise Pruning*, *c) Loop-wise Pruning*, and *d) Bit-wise Pruning*. In each stage as depicted in Figure 7.1, black parts represent the selected fault sites while the gray parts represent the pruned ones.

In the first stage, we perform *a) thread-wise pruning* where kernel threads are classified into different groups. This classification is based on the distribution of fault injection outcomes: threads in the same group share a similar application error resilience profile. From each group, we are able to randomly select *one thread* as the group representative. Yet, thread classification is challenging. In Section 7.3.2, we show that the *dynamic in-*

struction (DI) count per thread can be used as proxy for effective thread classification. We classify threads based on their dynamic instruction count into several groups, then select one representative (i.e., one black thread) per group.

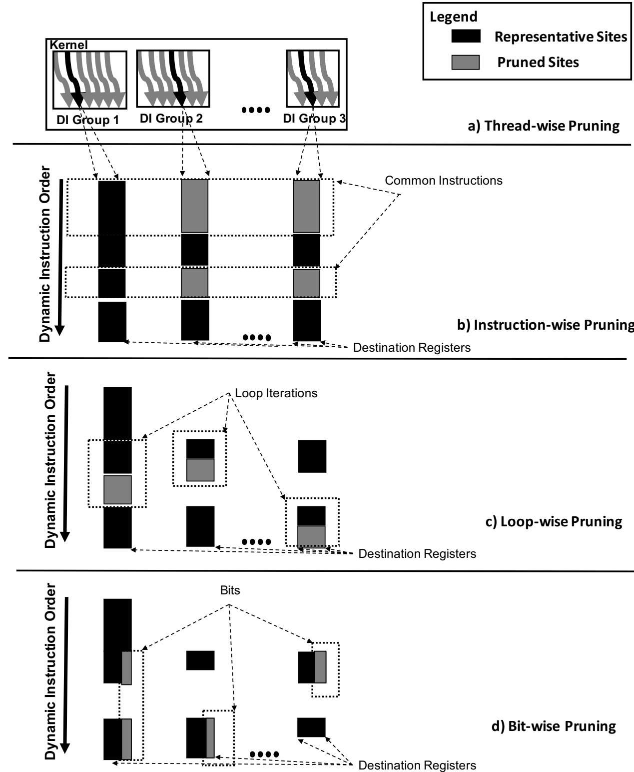


Figure 7.1: Overview of the 4-stage Fault Site Pruning Mechanism.

In the next pruning stage, we perform *b) instruction-wise pruning*, which leverages common blocks of code that are shared among the selected representative threads of the previous pruning stage. We find that because of the SIMT nature of the GPU execution model many threads execute the same subsets of instructions. These common instruction blocks are likely to have similar resilience characteristics (discussed further in Section 7.3.3), thus become candidates for pruning (see gray segments in Figure 7.1, stage b) Instruction-wise Pruning). Black segments are selected for fault injection and move to the next pruning stage.

In the subsequent pruning stage, *loop-wise pruning*, we identify loops in the threads

that are selected from the previous stage and we randomly sample several loop iterations to represent the entire loop block (we elaborate on how we do this sampling in Section 7.3.4). Within each loop, we are able to use a part of representative iterations (marked as black) and discard the rest (marked as gray), see Figure 7.1 stage c.

As a last step, with *bit-wise pruning*, we consider several pre-selected bit positions for fault injection. These bit positions are selected to cover a range of positions in registers to further reduce the fault site space (Section 7.3.5 gives the rationale behind the bit position selection). Similarly, to the rest of Figure 7.1, black bit positions are the selected fault sites while gray ones are pruned. Overall, Figure 7.1 gives a road-map of the progressive pruning steps that are discussed in detail in the following subsections.

To clarify that, in the rest of this section, the proposed pruning framework is illustrated with the results of a single input. For a different input, we have to follow the above four steps again to determine the pruned fault space. For example, the selected representative threads might be different for another input. The reason is that those pruning features are application-dependent and input-dependent. Fortunately, the profiling cost for one input is affordable as all information can be collected through one fault-free execution. Still it would be interesting to study the impact of different inputs on those pruning features, such as whether different inputs result in different CTA-wise and warp-wise grouping. Such insights (i.e., trends and variations) would be meaningful for developing efficient fault injection framework for different inputs.

7.3.2 Thread-Wise Pruning

As discussed in Section 7.2, GPGPU applications typically spawn thousands of threads. Therefore, injecting faults to all thread registers is not practical. To this end, we classify threads into groups that share similar resilience behavior. The challenge here is to choose an effective metric that can be easily extracted from the application to guide this classification.

In order to develop a classification process, we study the error resilience characteristics of CTAs and threads of a kernel through a large fault injection campaign (i.e., over 2

million fault injection runs). We investigate the fault resilience features *hierarchically*, starting from CTA-, thread-, and instruction-level. Our analysis illustrates that:

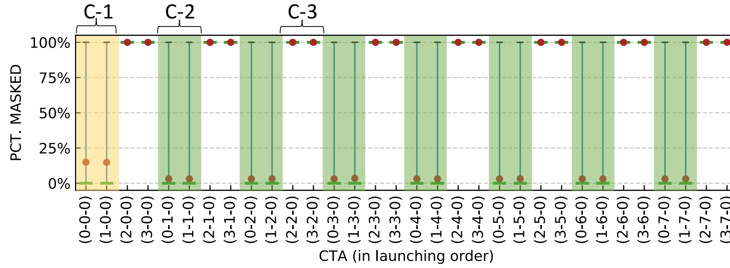
- A few representative CTAs and threads can capture the error resilience characteristics of the entire kernel.
- The number of dynamic instructions (short as *iCnt*) per thread can be used as an effective classifier to identify representative threads and guide the first pruning step.

7.3.2.1 CTA-wise Pruning

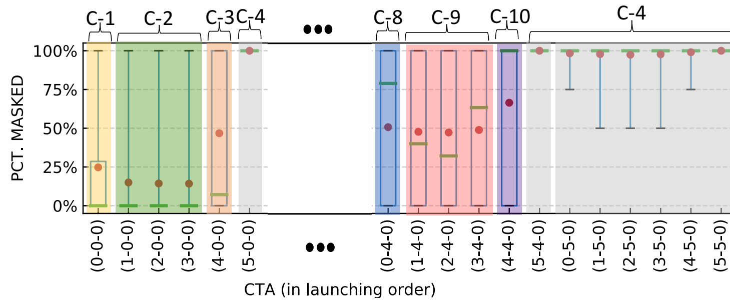
We first focus on understanding the error resilience characteristics at the CTA level. Although it is not practical to perform an exhaustive fault injection campaign at this level, it is relatively manageable to run exhaustive experiments for target instructions. We select a diverse set of dynamic instructions including memory access (e.g., *ld*), arithmetic (e.g., *add* and *mad*), logic (e.g., *and* and *shl*), and special functional instructions (e.g., *rcp*), and from different code locations (e.g., beginning, middle, and end). Although the fault sites are already reduced by targeting certain instructions and narrowing down to few locations, the number of (reduced) fault sites per kernel is still large, e.g., $1,217K$ for HotSpot, $774K$ for 2DCONV, $412K$ for K-Means.

Instead, we resort to Equation 7.4 to obtain n=$60K$ random samples for every target instruction in a kernel. We use 2DCONV and HotSpot, which are diverse nature in terms of number of threads and similarity across threads. For each application kernel, we manually select 5 instructions that cover the aforementioned diversity, resulting in $300K$ fault injection runs per application kernel. Figure 7.2(a)-(b) shows the grouping results given by one target instruction for 2DCONV and HotSpot, respectively. The results for the remaining four target instructions are not shown for brevity.

Figure 7.2(a) shows the distribution of fault injection outcomes for all 32 CTAs in 2DCONV. CTAs are listed in the order of their launching time along the x-axis. For every CTA, we calculate the percentage of *masked* outputs (percentage of *SDC* and *other*



(a) 2DCONV (line=34, opcode=*mad*)



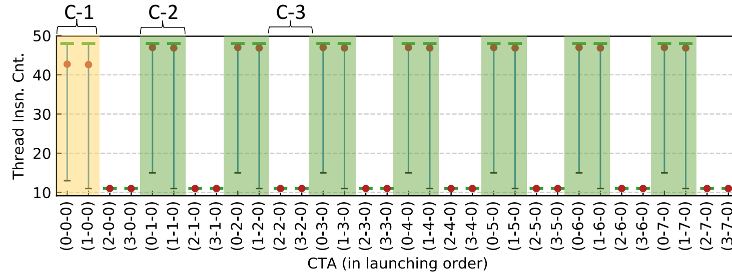
(b) HotSpot (line=52, opcode=*add*)

Figure 7.2: CTA grouping after $60K$ fault injection runs of one target instruction for (a) 2DCONV and (b) HotSpot. CTAs with the same color are classified into the same group. In the box plot, the horizontal green lines represent the median and red dots represent the mean.

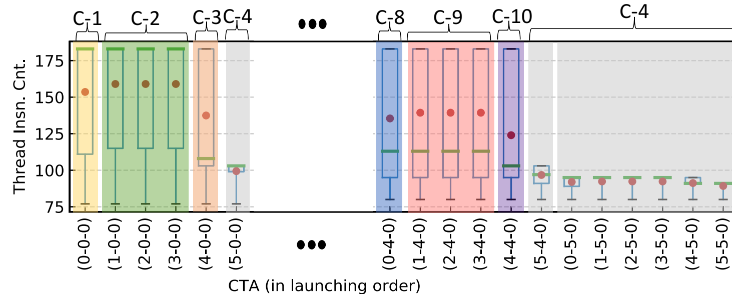
outputs are not shown) for each of its 256 threads and show the distribution of *masked* outputs using boxplots (i.e., one boxplot for each CTA to illustrate salient points in the distribution of masked outputs, including the 25th and 75th percentiles, and the mean and median). We observe that CTAs exhibit three distinct distributions as given by the different shapes of boxplots. Each group is marked by a different color. Therefore, 3 CTAs (one per group) is sufficient to represent the entire kernel. Similarly, Figure 7.2 (b) shows the CTA grouping results for HotSpot. There are 36 CTAs in total, each containing 256 threads. For clarity, we show a few CTAs only. We observe that HotSpot has more diverse CTAs than 2DCONV and hence we classify its CTAs into 10 groups (C-1 to C-10).

Although the experiments illustrated in Figure 7.2 point to a promising methodology to obtain a first-order CTA grouping, it is obtained with $300K$ fault injection runs per kernel. This is still not always practical, as one can always opt to the random fault injection campaign [83], which requires $60K$ runs. Therefore, it is imperative to find an effective

metric that can further prune the fault space. We show that the number of dynamic instructions per thread (*iCnt*) is an alternative good measure for thread classification. This is encouraging as only *one* fault-free execution is sufficient to collect all the required *iCnt* information.



(a) 2DCONV



(b) HotSpot

Figure 7.3: CTA grouping given by average dynamic thread instruction count (*iCnt*) per CTA for (a) 2DCONV and (b) HotSpot. CTAs with the same color are classified into the same group. A significant similarity is is observed with Figure 7.2.

Figure 7.3(a)-(b) shows the results for 2DCONV and Hotspot. Each boxplot shows the distribution of thread *iCnt* per CTA. Recall that each boxplot in Figure 7.2 represents the distribution of percentage of *masked* outputs. Similarly here, we are able to classify the CTAs into the same groups as in Figure 7.2 (both Figure 7.2 and 7.3 use the same color-code). Table 7.3 and 7.4 report the grouping results guided by the average thread *iCnt* per CTA (given by Figure 7.3) for 2DCONV and HotSpot, respectively (see the left three columns).

To summarize, the above results confirm that *iCnt* is effective in capturing the error

Table 7.3: CTA and threads groups for 2DCONV.

CTA Grp.	Avg. iCnt	CTA Proportion	Thd. Grp.	Thd. iCnt	Thd. Proportion*
C-1	43	6.25%	T-11	13	12.50%
			T-12	15	2.73%
			T-13	48	84.77%
C-2	47	43.75%	T-21	15	3.13%
			T-22	48	96.87%
C-3	11	50.00%	T-31	11	100.00%

* For each CTA group, we show its percentage of threads belonging to the corresponding thread group.

resilience characteristics at the CTA-level. Based on the grouping guided by *iCnt*, only a few CTAs per kernel are sufficient to capture the entire picture. We have conducted similar experiments for other application kernels that overwhelmingly support the above conclusion.

Observation 7.1 *A few CTAs are enough to capture the error resilience characteristics of a kernel. These CTAs are selected based on the average thread dynamic instruction count (iCnt).*

7.3.2.2 Thread-wise Pruning

By narrowing down to only a few CTAs in a kernel, we are able to significantly reduce the number of fault sites. Yet, an exhaustive fault injection campaign using all threads in selected CTA representatives is not viable. For example, for a CTA with 256 threads, if each thread executes an average of 100 dynamic instructions and if all destination registers are 32-bit wide, then a total of $819,200$ runs are needed. Therefore, we continue the thread classification within each CTA in order to select only a few representative threads. As done previously, we classify threads inside a CTA using (1) a large number of fault injection runs and (2) *iCnt*. We confirm that the two methods lead to the same thread grouping results, see Figure 7.4. In other words, thread *iCnt* is also effective within a CTA to classify

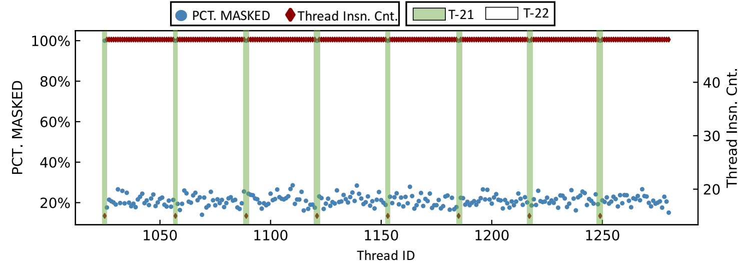
Table 7.4: CTA and threads groups for HotSpot.

CTA Grp.	Avg. iCnt	CTA Proportion	Thd. Grp.	Thd. iCnt Range	Thd. Proportion*
C-1	154	2.78%	T-11	77 – 98	23.44%
			T-12	111 – 115	10.55%
			T-13	183	66.02%
C-2	159	8.33%	T-21	77 – 90	12.50%
			T-22	108 – 115	16.41%
			T-23	183	71.09%
C-3	137	2.78%	T-31	77 – 103	45.31%
			T-32	108 – 115	8.98%
			T-33	183	45.70%
C-4	99	30.56%	T-41	77 – 99	28.91%
			T-42	103	71.09%
C-5	160	8.33%	T-51	89 – 111	18.75%
			T-52	113	5.08%
			T-53	115	5.08%
			T-54	183	71.09%
C-6	166	25.00%	T-61	108	6.25%
			T-62	111	6.25%
			T-63	113 – 115	10.94%
			T-64	183	76.56%
C-7	143	8.33%	T-71	95 – 108	43.75%
			T-72	113 – 115	7.03%
			T-73	183	49.22%
C-8	135	2.78%	T-81	80 – 98	45.31%
			T-82	111 – 113	8.98%
			T-83	183	45.70%
C-9	139	8.33%	T-91	80 – 95	37.50%
			T-92	108 – 113	13.28%
			T-93	183	49.22%
C-10	124	2.78%	T-101	80 – 103	60.94%
			T-102	108 – 113	7.42%
			T-103	183	31.64%

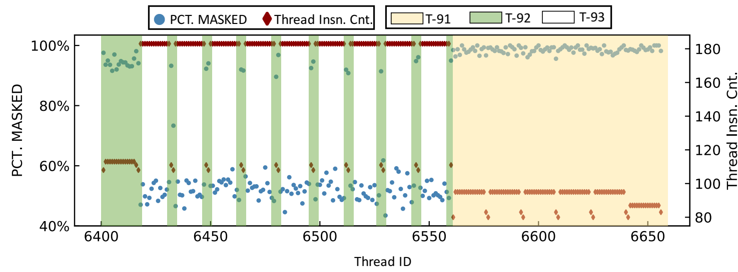
* For each CTA group, we show its percentage of threads belonging to the corresponding thread group.

threads.

Figure 7.4(a) shows results for 2DCONV. Each blue dot represents the percentage of *masked* outputs in that thread (left y-axis) and each red dot indicates the corresponding



(a) 2DCONV: CTA Group C-2



(b) HotSpot: CTA Group C-9

Figure 7.4: Thread Grouping inside one CTA.

thread *iCnt* (right y-axis). We mark threads in the same group with the same color. We observe a clear repeating pattern that allows for classifying all threads into two distinct groups (one marked with green color, the other one is uncolored, see Figure 7.4(a)):

1. T-21: threads with *iCnt*=15 and percentage of *masked* outputs at around 100%.

2. T-22: threads with *iCnt*=48 and percentage of *masked* outputs between 20% to 30%.

Table 7.3 reports the thread grouping details for 2DCONV (right three columns). A potential reason for such similarity in the distribution of fault injection outcomes among threads with different *iCnt* is the fact that these threads share large common code blocks, this is further discussed in Section 7.3.3.

Figure 7.4(b) shows that threads in HotSpot can be also classified into several groups (Table 7.4). Due to the complexity of this kernel, we merge thread groups with similar *iCnt* together for visualization purposes, resulting in 3 distinct groups: one marked in green, one marked in yellow, and the third one is uncolored. Note that, during the actual fault

injection campaign, we still classify threads based on the exact thread *iCnt* (a total of 87 thread groups across selected CTAs) and select one representative thread per group.

We find that it is important to perform the grouping in two steps: first at the CTA level and then at the thread level. Through our fault injection runs, we find that threads with the same *iCnt* from different CTAs could have different instructions and thus show different distribution of fault injection outcomes (this is observed in HotSpot and Gaussian K2). Therefore, the step of CTA-wise grouping cannot be skipped.

Observation 7.2 *Threads can be further classified within a CTA. A few threads within a CTA are able to represent the CTA's error resilience characteristics.*

7.3.3 Instruction-Wise Pruning

Our analysis shows that different threads normally share a large portion of common instructions. We aim to further prune the fault sites by finding common instruction blocks among the resulted set of thread representatives after the *thread-wise pruning* stage. We illustrate this observation using PathFinder application. Figure 7.5 shows instruction snippets of its two representative threads (“a" and “b") chosen from the previous pruning stage. Comparing their PTXPlus code, dynamic instructions from the first line till line number 53 are all the same; thread “a" has 17 more instructions in the middle; at the end, all the remaining 463 instructions across the two threads are also the same.

Table 7.5 shows the percentage of *masked* and *SDC* outputs for PathFinder *if soft errors are injected in their common portion only.* The distributions of fault injection outcomes that stem from this common block are quite close (see columns 4 and 5 in the table). Naturally, fault injections have to occur in the entire body of thread “a" to calculate its resilience, but since there is a common code block across the two threads, it can be used to extrapolate the distribution of fault injection outcomes of thread “b". This eliminates the need to inject faults in thread “b" and essentially prunes the fault sites generated for this thread. We introduce -0.078% error for the percentage of *masked* outputs and -0.031%

Thread “a” ($iCnt = 533$)

```
1   shl.u32 $r3, s[0x0010], 0x00000001
2   cvt.u32.u16 $r1, %ctaid.x
3   add.u32 $r3, -$r3, 0x00000100
4   mul.wide.u16 $r4, $r1.lo, $r3.hi
5   mad.wide.u16 $r4, $r1.hi, $r3.lo, $r4
    ......
49  cvt.s32.s32 $r2, -$r2
50  and.b32 $p0|$o127, $r5, $r2
51  ssy 0x00000228
52  mov.u32 $r2, $r124
53  @$p0.eq bra l0x00000228
54  add.half.u32 $r7, s[0x0038], $r1
55  mov.half.u32 $r2, s[0x0030]
56  mul.wide.u16 $r8, $r2.lo, $r7.hi
57  mad.wide.u16 $r8, $r2.hi, $r7.lo, $r8
58  shl.u32 $r8, $r8, 0x00000010
    ......
66  min.s32 $r7, s[$ofs2+0x0040], $r8
67  ld.global.u32 $r2, [$r2]
68  add.u32 $r2, $r2, $r7
69  mov.u32 s[$ofs3+0x0440], $r2
70  mov.u32 $r2, 0x00000001
71  l0x00000228: nop
72  bar.sync 0x00000000
73  set.eq.s32.s32 $p0/$o127, $r6, $r1
74  @$p0.ne bra l0x000002b8
75  set.ne.s32.s32 $p1/$r1, $r2, $r124
    ......
529 set.eq.s32.s32 $p0/$o127, $r6, $r1
530 @$p0.ne bra l0x000002b8
531 l0x000002b8: set.ne.s32.s32 $p0/$o127, $r2,
    $r124
532 bra l0x000002c8
533 l0x000002c8: @$p0.eq retp
```

Thread “b” ($iCnt = 516$)

```
1   shl.u32 $r3, s[0x0010], 0x00000001
2   cvt.u32.u16 $r1, %ctaid.x
3   add.u32 $r3, -$r3, 0x00000100
4   mul.wide.u16 $r4, $r1.lo, $r3.hi
5   mad.wide.u16 $r4, $r1.hi, $r3.lo, $r4
    ......
49  cvt.s32.s32 $r2, -$r2
50  and.b32 $p0|$o127, $r5, $r2
51  ssy 0x00000228
52  mov.u32 $r2, $r124
53  @$p0.eq bra l0x00000228
54  l0x00000228: nop
55  bar.sync 0x00000000
56  set.eq.s32.s32 $p0/$o127, $r6, $r1
57  @$p0.ne bra l0x000002b8
58  set.ne.s32.s32 $p1/$r1, $r2, $r124
    ......
512 set.eq.s32.s32 $p0/$o127, $r6, $r1
513 @$p0.ne bra l0x000002b8
514 l0x000002b8: set.ne.s32.s32 $p0/$o127, $r2,
    $r124
515 bra l0x000002c8
516 l0x000002c8: @$p0.eq retpz
```

Figure 7.5: PTXplus code comparison of two representative threads for PathFinder. Blue bold lines indicate common instructions.

error for the percentage of *SDC* outputs (both minimal variations), but with a significant reduction of 12,344 fault sites.

Table 7.5: Effect of instruction-wise pruning for two threads.

Application	Thread	% Common Insn.	% MSK	% SDC
PathFinder	a	92.1%	89.4%	0.0%
	b	100.0%	90.1%	0.4%

To confirm that this behavior persists across kernels, we conduct exhaustive experiments across the fault site space after *CTA-wise* and *thread-wise* pruning and confirm that common blocks of instructions across threads share a surprisingly similar distribution of fault injection outcomes (Table 7.6). The third column of Table 7.6 shows the percentage of pruned common instructions, and the 4th and 5th columns show the error of pruned results, compared to the exhaustive experiments before pruning common instruction blocks. This

pruning technique is useful for complicated applications such as PathFinder and HotSpot, with the reduction of 92.81% instructions and 92.80% instructions, respectively. Table 7.6 shows that the percentage of common instructions pruned in applications kernels ranges from 42.86% to 92.81% and the error introduced by pruning common instruction blocks for *masked* and *SDC* outputs is −0.15% and −0.1%, respectively.

Table 7.6: Summary of instruction-wise pruning for selected kernels. Other kernels do not exhibit instruction commonality.

Application	**Kernel**	**% Pruned Common Insn.**	**Introduced Error**	
			MSK	**SDC**
HotSpot	K1	92.81%	-0.14%	0.14%
PathFinder	K1	92.80%	0.03%	-0.09%
LUD	k46	80.00%	-0.78%	-0.70%
2DCONV	k1	66.67%	0.09%	-0.09%
Gaussian	K2	62.50%	-0.13%	0.13%
Gaussian	K126	42.86%	0.00%	0.00%
Average		72.94%	-0.15%	-0.10%

Note that several application kernels (e.g., 2MM, MVT, SYRK, and GEMM) after thread-wise pruning end up with only one representative thread. These kernels are not suitable for instruction-wise pruning, and are therefore not included in the table. For Gaussian K1 and K2, and K-Means K1, instruction-wise pruning is also not applicable. For these application kernels, there are two representative threads, one with very few instructions (i.e., less than 10) and other with many (i.e., hundreds or thousands), leaving few opportunities to explore code commonality.

Observation 7.3 *Different representative threads may share significant portions of common instructions. Therefore, distributions of fault injection outcomes of these common portions are similar. Consequently, a large number of fault sites can be pruned while achieving significant accuracy.*

7.3.4 Loop-Wise Pruning

Table 7.7 shows the total number of instructions and the number of loop iterations per kernel. The kernels are sorted in increasing order by the portion of instructions in loops (after the loop is unrolled). Excluding kernels with no loops, a large portion of instructions in a kernel come from loop iterations, ranging from 65.79% in LUD K46 to 99.71% in MVT. Such an abundance in the repetitive instruction blocks indicates large opportunities for pruning. We aim to discover whether the distribution of fault injection outcomes can be captured by a subset of loop iterations.

Table 7.7: Statistics related to loops.

Application	**Kernel**	**# Thd.**	**# Loop Iter.**	**% Insn. in Loop**
HotSpot	K1	9216	0	0.0%
2DCONV	K1	8192	0	0.0%
NN	K1	43008	0	0.0%
Gaussian	K1	512	0	0.0%
	K2	4096	0	0.0%
	K125	512	0	0.0%
	K126	4096	0	0.0%
LUD	K45	256	0	0.0%
	K46	16	120	65.79%
	K44	32	120	78.75%
K-Means	K1	2304	34	82.42%
	K2	2304	170	87.6%
PathFinder	K1	1280	20	92.84%
SYRK	K1	16384	128	98.13%
2MM	K1	16384	128	98.18%
GEMM	K1	16384	128	98.21%
MVT	K1	512	512	99.71%

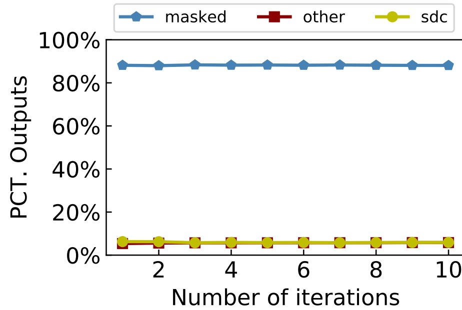
Towards this goal, we consider a number of randomly sampled iterations for fault injections. We present results for different fault site sizes, defined by the total number of sampled iterations (num_iter) ranging from 1 to 15. Figure 7.6 shows the impact of num_iter on the distribution of fault injection outcomes for PathFinder, SYRK, and K-Means K1. For K-Means K1, we show the effect of two different random seeds for sampling

the loop iterations. We observe that the distribution of fault injection outcomes is stable after a certain number of sampled loop iterations. Looking closer into the application source code, we observe that: 1) several loop conditions are controlled by constants and not variables that are changed within the loop and 2) there is no data communication among different loop iterations. Therefore, there is no error propagation among different loop iterations, thus sampling is sufficient for obtaining the distribution of fault injection outcomes. These observations hold true for the evaluated applications, but may not be true for other applications.

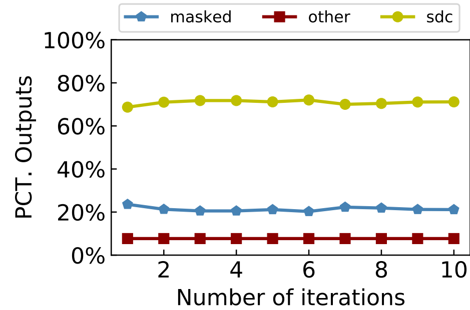
Figure 7.6 shows that different applications require different numbers of sampled loop iterations to reach stability for the percentage of masked, SDC, and other outputs. Figure 7.6(a) shows that PathFinder requires 3 sampled loop iterations. Figure 7.6(b) shows that the output of SYRK becomes stable after 8 sampled loop iterations. In both cases the trend is clear. For K-Means K1 (Figure 7.6(c)), there is no clear trend with a few sampled iterations but results stabilize when the number of sampled loop iterations reaches 15. To further explore the behavior of this kernel, we sample the loop iterations of K-Means K1 using another random seed. Figure 7.6(d) reports the results and shows that stability is again achieved with 15 loop iterations, as shown in Figure 7.6(c).

To summarize, Figure 7.6 suggests that randomly sampling a few iterations is generally sufficient in capturing the distribution of fault injection outcomes of application kernels. This offers another way to further reduce the fault sites within a thread. Similar experiments are done for all other applications and result in the same conclusion. Therefore, we randomly add iterations one by one, until the result is stable. For the examined kernels, the number of iterations sampled among loops differs from a minimum of 3, to a maximum of 15, with an average of 7.22 iterations across all application kernels.

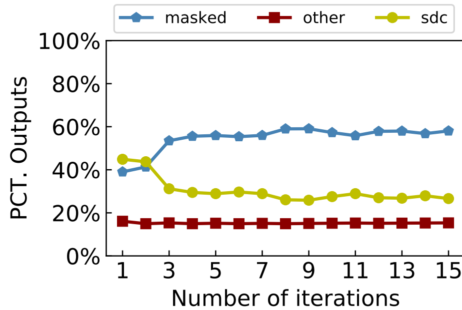
Observation 7.4 *Distribution of fault injection outcomes in a kernel can be captured by a subset of iterations in the loop. This provides an opportunity for fault site pruning thanks to the abundance of instructions in a loop.*



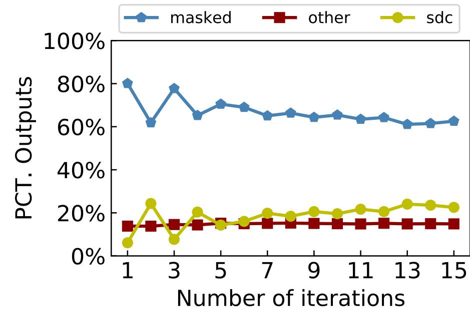
(a) PathFinder (Max # of Loop Iterations=20)



(b) SYRK (Max # of Loop Iterations=128)



(c) K-Means K1 (Max # of Loop Iterations=34)



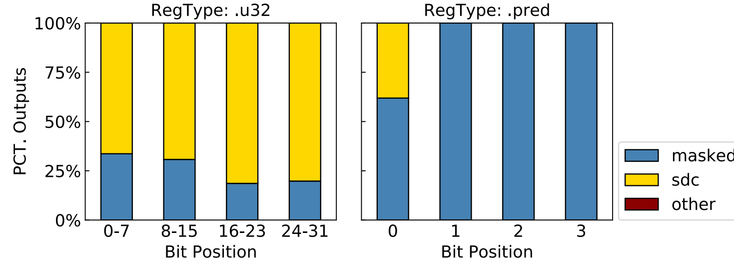
(d) K-Means K1, using a different seed

Figure 7.6: Impact of loop-wise pruning on distribution of fault injection outcomes for (a) PathFinder, (b) SYRK, and (c)-(d) for K-Means K1 with different random seeds.

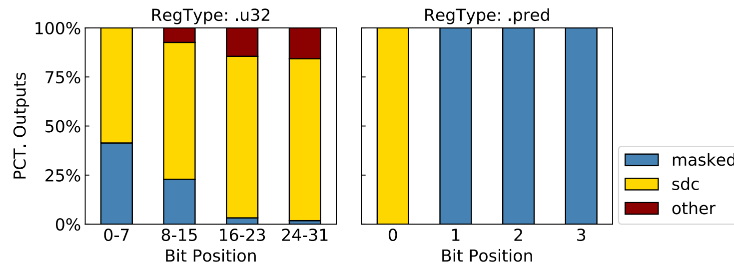
7.3.5 Bit-Wise Pruning

Beyond instruction-wise pruning, we explore whether it is possible to further prune the fault site space from the perspective of bit positions. The intuition is that not all bit positions contribute equally to incorrect outputs. Intuitively, one may assume that bit flips in higher bit positions would produce more problematic outputs as the difference between the original value and flipped value tends to be larger. However, this intuition does not always hold true. The error pattern depends on application kernels and register types.

Figure 7.7(a)–(b) presents the distribution of fault injection outcomes for two major types of registers (i.e., *.u32* and *.pred*) for 2DCONV and MVT, respectively. We evenly



(a) 2DCONV



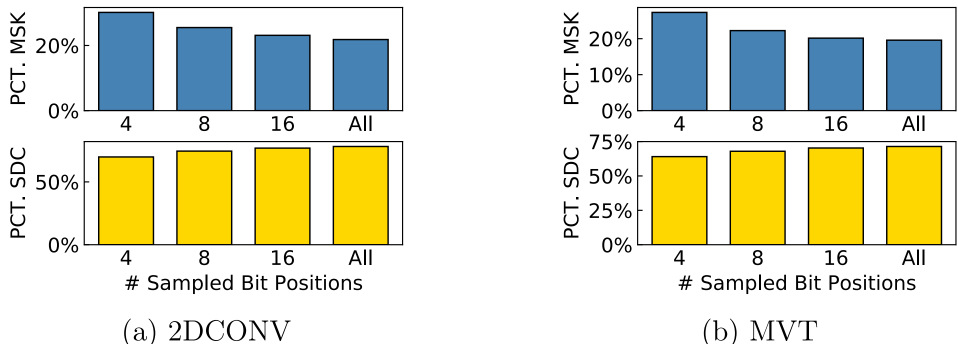
(b) MVT

Figure 7.7: Distribution of fault injection outcomes of different bit position sections of two major register types (*.u32* and *.pred*) for (a) 2DCONV and (b) MVT.

partition bit positions in a register into 4 sections and show the distribution of fault injection outcomes for every section. First, we notice that for register type *.u32*, the intuition of higher bit sections having more problematic outputs holds for both application kernels. For MVT, the percentage of *masked* outputs decreases with increasing bit positions and becomes almost invisible in the higher two bit sections. For register type *.pred* that has 4 bits, we observe that for both applications, the lowest bit position results in output errors, while the higher three bit positions are very error resilient (they result only in *masked* outputs). This is the nature of 4-bit predicate system [5]: the highest three bits in register type *.pred* are used for overflow flag, carry flag, and sign flag, respectively, while the lowest bit represents the zero flag. Within the context of the applications we study in this work, only the zero flag is used for branch conditions, so we can confidently prune the other three bit positions in register type *.pred*.

Note that since the *.pred* register is not a common one, the scope of pruning is not significant. For *.u32* (see Figure 7.7) there is a consistent pattern as a function of the bit

position, therefore we select several bit positions from each register section resulting in a total of 4, 8, and 16 bit samples (at most, depending on the register size) and compare the distribution of fault injection outcomes with that of all bit positions. Note that the selected bits are separated by equal intervals. For instance, for a 32-bit register and selecting 2 bit positions per section, we focus on bits in the following positions $\{3, 7, 11, 15, 19, 23, 27, 31\}$. Figure 7.8 shows the results. For 2DCONV (see Figure 7.8 (a)), the change in distribution of fault injection outcomes changes as the number of sampled bits increase. This behavior persists in Figure 7.8 (b) for MVT. Overall, sampling 16 bits is promising as fault site space can be significantly pruned.



(a) 2DCONV (b) MVT

Figure 7.8: Impact of bit-wise pruning on distribution of fault injection outcomes for (a) 2DCONV and (b) MVT (all registers). Percentage of outputs stabilizes at 16 bits.

Observation 7.5 *It is possible to reduce the number of fault sites by examining only a subset of bit positions.*

7.4 Evaluation

In this section, we evaluate the proposed progressive pruning methodology by comparing with $60K$ random experiments (baseline case, see Section 7.2.3).

We calculate the distribution of fault injection outcomes for every application kernel and compare with the percentage numbers given by the *baseline* (the closest approximation to ground truth as discussed in Section 7.2.1). The error margin and confidence interval of

baseline are set to 0.63% and 99.8%, respectively. Figure 7.9 shows the comparison results. We observe that our pruning method produces very accurate error resilience estimations for several benchmark kernels including Hotspot, K-Means K2, Gaussian K2, Gaussian K126, PathFinder, LUD K44, LUD K46, 2DCONV, GEMM, and SYRK. For these kernels, the difference in terms of the percentage of *masked* outputs comparing with *baseline* is always less than 1%. For the remaining kernels, there is no significant mismatch from the *baseline*. On average, the differences in terms of *masked*, *SDC*, *other* outputs are 1.68%, 1.90%, and 1.64%, respectively.

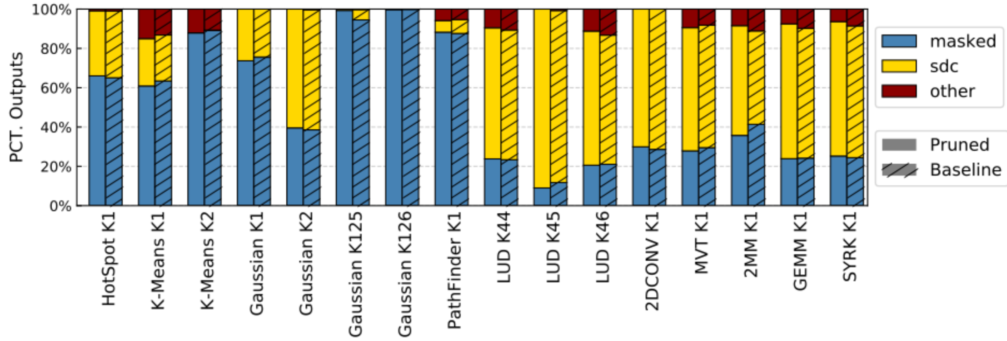
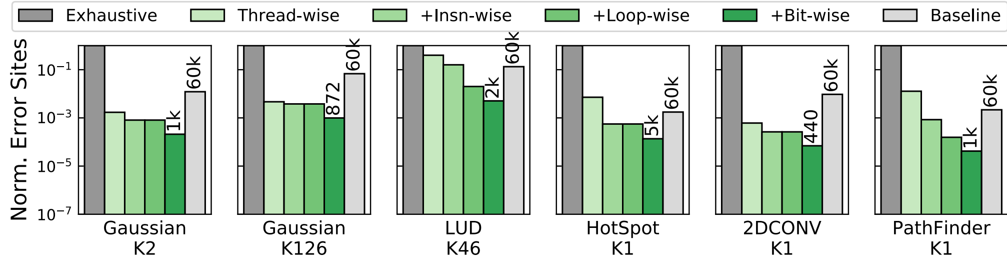
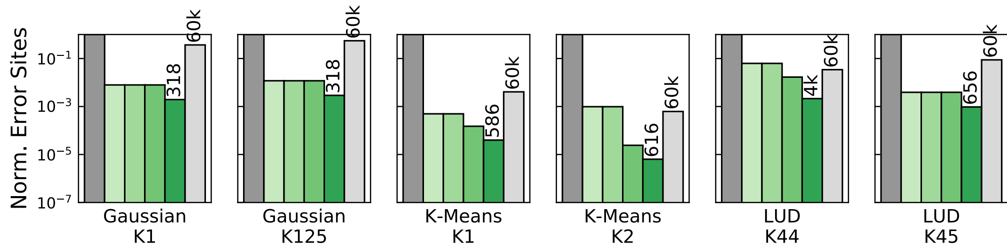


Figure 7.9: Error resilience comparison of progressive fault site pruning techniques against the ground truth (baseline).

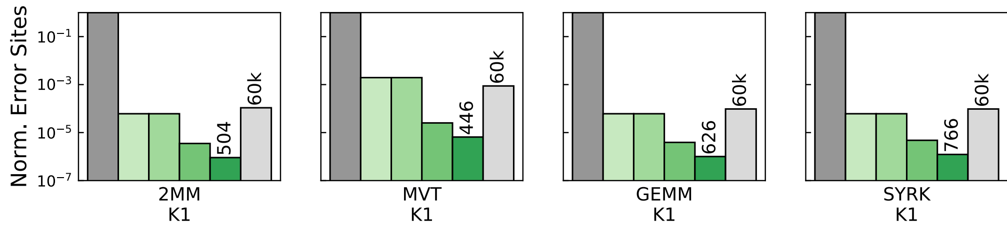
Next, we compare the effectiveness of the proposed progressive feature-based pruning in terms of fault site reduction. Figure 7.10 shows the comparison results. Note that we use log scale with a base of 10 for the y-axis. The number of fault sites left after each pruning step is normalized by the original exhaustive fault sites for every application kernel for cross-kernel comparison. The height of each bar represents the normalized number of fault sites after each step and the decrease in bar height from the previous bar indicates the reduction in fault site space. The last two bars in each sub-figure report also a number that indicates the fault site size of the fully pruned space versus the $60K$ baseline case which is the closest to the ground truth. Note that our pruning technique needs one-time offline profiling to collect the application features needed for pruning. We observe from Figure 7.10(a) that *Thread-wise pruning* is the most effective, as it reduces the magnitude



(a) Kernels with instruction-wise commonality



(b) Kernels without instruction-wise commonality



(c) Kernels not applicable to instruction-wise commonality

Figure 7.10: Fault site reduction comparison based on various feature-based pruning techniques. "+" indicates that each pruning stage is progressively built upon the pruned sites resulted from the previous stage. The height of the pruned fault sites bar is normalized by the original exhaustive fault sites for each application kernel, see last column of Table I. The effectiveness of progressive fault site pruning is compared against comprehensive baseline injection ($60K$ random experiments). The exact numbers are shown on the top of the last two columns for the proposed method and the baseline case, respectively.

of the number of fault sites by up to 5 orders of magnitude. With *Thread-wise pruning*, we only use a few representative threads (i.e., less than 10) per application kernel. This is a significant reduction compared to the original number of threads per kernel, e.g., 1 representative out of 16384 threads for GEMM, SYRK, and 2MM, and 6 representatives out of 8192 threads for 2DCONV. Such efficient first-order thread-wise pruning lays a substantial base for the following steps. One important clarification that needs to be stated

is that any later pruning is performed on the selected thread representatives, therefore further reductions after this step are expected to be modest.

Instruction-wise pruning exploits the commonality among the thread representatives selected in the previous step. It is important to clarify that kernels in the second row (see Figure 7.10 (b)) are not suitable for *Instruction-wise pruning*, because their representative threads do not have many common instruction blocks. Kernels in Figure 7.10 (c) are not applicable to *Instruction-wise pruning* as there is only one thread group per kernel, i.e., they only have a single representative thread. Comparing results within the first row of Figure 7.10, we observe that *Instruction-wise pruning* is most effective for HotSpot and PathFinder, with the reduction of 92.81% and 92.80% instructions, respectively.

Loop-wise and *Bit-wise pruning* progressively contribute to the reduction of the fault sites for each application kernel. The effectiveness of *Loop-wise pruning* depends on the percentage of loop instructions in the fault sites left by the previous step. We observe a large reduction in K-Means K2, LUD K46 and matrix-related applications including 2MM, GEMM, SYRK, and MVT. This matches the fact that there is a large portion of loop instructions in these kernels (see Table 7.7). On the other hand, the effectiveness of *Bit-wise pruning* is relative stable, i.e., the percentage of reduction in fault sites obtained by *Bit-wise pruning* is consistent across kernels.

Summary: We present results of the 10 applications (16 kernels) using the pruned fault site subspaces outlined above to seek the distribution of application outputs (*masked*, *SDC*, and *other*). Our proposed mechanism is able to produce comparable distribution numbers of fault injection outcomes against a comprehensive baseline injection of $60K$ experiments which we use here as a statistically sound approximation of ground truth. For each step of feature-based progressive fault site pruning, we observe significant progressive reduction in the number of fault sites, ending up with only a few hundreds of fault sites in several kernels.

7.5 Multi-Bit Fault Injection

In the previous sections, we present and evaluate a progressive fault site pruning technique, which is able to accurately estimate the distribution of fault injection outcomes but with a much smaller number of fault-injection runs. In this technique, we focus on injecting single bit fault per run. In addition, it is also interesting to see how multiple single-bit faults affect the outcome of application runs. Multi-bit faults in this context represent more than one bit flips that occurred during the application run on different threads, i.e., one single-bit fault per thread.

In the rest of this section, we first clarify the assumptions we make to extend the single-bit fault site pruning technique to the context of multi-bit faults. Then, we discuss how to obtain the distribution of multi-bit faults injection outcomes using the single-bit fault injection outcomes. Lastly, we evaluate the pruning accuracy in the context of injecting multiple faults and present observations on the error resilience characteristics of applications over increasing number of injecting faults.

7.5.1 Assumptions

In this subsection, we state several assumptions to extend the result of single-bit fault injection to the context of multi-bit faults. Note that the goal here is *not* to provide the most accurate estimation of the outcomes of multi-bit fault injection runs, but how to take advantage of the distribution of single-bit fault injection to estimate the output distribution in the presence of multiple single-bit faults. We make the following two assumptions.

Assumption-1 *We consider injecting multi-bit faults into different threads for GPGPU applications.*

When injecting multiple faults in a single application run, those faults can be injected a) in the same register, b) in different registers but in the same thread, and c) in different threads. Sangchoolie et al. [124] study the impact of multi-bit faults for CPU applications under the context of scenarios a) and b) and conclude that one bit is often enough. They

do not consider scenario c) because CPU applications are normally single threaded. In contrast, GPGPU applications can spawn hundreds to thousands threads (see Table 7.1), making scenario c) a significant one. Here, we focus on scenario c) in the context of GPGPU applications, which is injecting multi-bit faults in different threads throughout the application run.

Assumption-2 *We assume that threads do not interact with each other.*

In current GPGPU programming, threads can communicate through shared memory [8, 4, 10]. Intuitively, since GPGPU applications aim at exploiting a high parallelism degree, there should be very little shared memory communication otherwise latency becomes high again. To validate this intuition, we look into the source codes of the benchmarks selected in this chapter (see Table 7.1) and make the following observations:

1. Most benchmarks do not use shared memory at all, including all benchmarks in the Polybench suite and Gaussian Elimination in the Rodinia suite.

2. Only a few benchmarks use shared memory. However, we find that they use shared memory to efficiently read input and write output, rather than for the purpose of thread communication. Benchmarks including HotSpot, K-Means and PathFinder fall in this category.

3. LU Decomposition is the only benchmark that does not fall under the previous two categories. In this benchmark, partial kernels (i.e., *lud_perimeter* and *lud_diagonal*) use shared memory for thread communication. For this case, we are able to provide only an upper bound of the error resilience level, because if threads communicate with each other, then the aftermath of multi-bit faults should be the same or worse as compared to no communication among threads.

7.5.2 Extending Pruning to Multi-bit Fault Injection

So far, we have stated the two assumptions that we can use to simplify the complexity of multi-bit fault injection. Here, we explain how to take advantage of the results of single-

bit fault injection to obtain the outcome of multi-bit fault injection. For the remaining of this chapter, we consider injecting multi-bit faults into different threads and ignore thread communication (if any) for GPGPU applications.

7.5.2.1 Extending to double-bit fault injection

We start with the scenario of considering two faults. Injecting two faults is actually a two-step procedure. We first select one fault site either randomly (i.e., the baseline technique) or as instructed by the progressive fault sites pruning technique (see Section 7.3). If the first fault does not cause the program to crash, then we can inject a second fault after the first one till the end of the execution, still following the same manner. Given that the distribution of single-bit fault injection outcomes of one benchmark kernel is $x\%$ *masked*, $y\%$ *SDC*, and $z\%$ *other*, which is obtained by either the baseline technique (i.e., 60K experiments) or the fault sites pruning technique. The outcome of two-bit fault injection can be then calculated as follows:

1. $masked\% = x\% \times x\%$,
2. $SDC\% = x\% \times y\% + y\% \times x\% + y\% \times y\%$,
3. $other\% = 1 - masked\% - SDC\%$.

7.5.2.2 Extending to multi-bit fault injection

We then extend the procedure to obtain the outcome of m-bit fault injection recursively by adding one single-bit fault to the outcome of injecting m-1 faults if the m-1 faults do not cause the program to crash. Given that the distribution of the m-1 faults injection outcome of one benchmark kernel is $x_{m-1}\%$ *masked*, $y_{m-1}\%$ *SDC*, and $z_{m-1}\%$ *other* and that the distribution of single fault injection outcome is $x_1\%$ *masked*, $y_1\%$ *SDC*, and $z_1\%$ *other*, then we can calculate the outcome of m-bit fault injection as follows:

1. $masked\% = x_{m-1}\% \times x_1\%$,

2. $SDC\% = x_{m-1}\% \times y_1\% + y_{m-1}\% \times x_1\% + y_{m-1}\% \times y_1\%$,

3. $other\% = 1 - masked\% - SDC\%$.

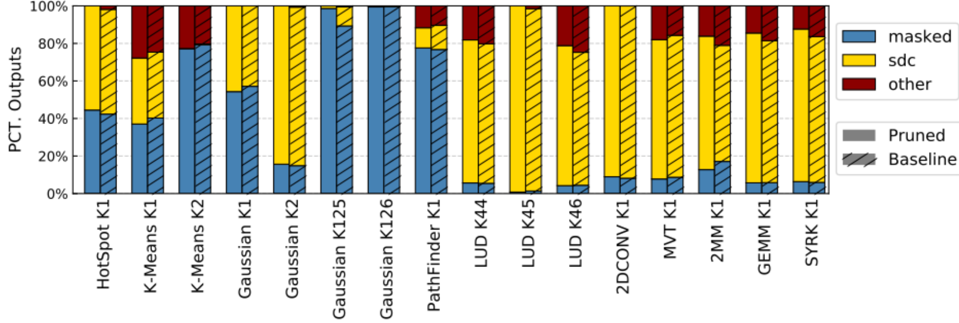
7.5.3 Evaluation

In this section, we evaluate the results of multi-bit fault injection. We first compare the discrepancy of extending to multi-bit fault injection using the outcome of single-bit fault injection obtained by the baseline technique (i.e., the 60K random experiments, which is closest to the ground truth) and the proposed progressive fault site pruning technique (see Section 7.2.1). Then, we present how the error resilience characteristics change over an increasing number of injected faults for different benchmark kernels.

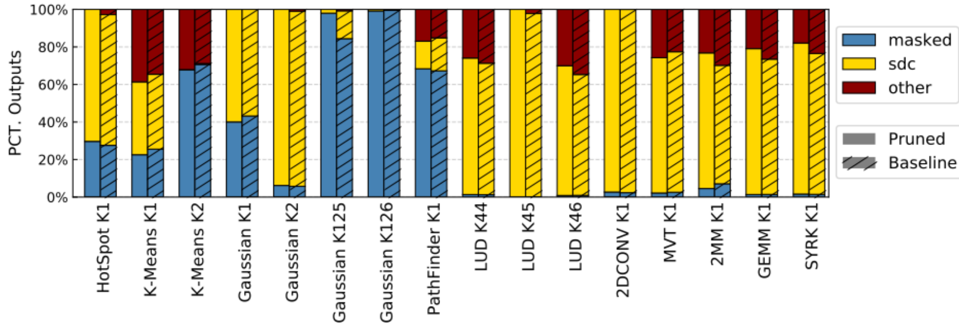
7.5.3.1 Comparison of accuracy

We start by comparing the outcomes obtained using the proposed progressive fault site pruning technique against the *baseline*, which is the closest approximation to ground truth as discussed in Section 7.2.1. The error margin and confidence interval of *baseline* are set to 0.63% and 99.8%, respectively. Figure 7.11(a) shows the distribution of double-bit fault injection outcomes for every application kernel. We observe that the pruning method still produces accurate estimations of error resilience for most of the benchmark kernels. For example, the difference in terms of the percentage of *masked* outputs is within ±1% for 10 out of the 16 selected benchmark kernels, including Gaussian K2, Gaussian K126, PathFinder, LUD K44, LUD K45, LUD K46, 2DCONV, MVT, SYRK, and GEMM. Gaussian K125 shows the largest difference (9%), which is also the most challenging one for the pruned space even in the single-bit fault injection campaign (see Figure 7.9). The differences in terms of the percentage of *SDC* and *other* outputs are slightly larger but still acceptable for most benchmark kernels. The number of benchmark kernels where the discrepancy is within ±1% is 6 and 5 for *SDC* outputs and *other* outputs,

respectively. On average, the differences in terms of the percentage of *masked*, *SDC*, and *other* outputs are 1.81%, 2.58%, and 2.03%, respectively.



(a) Outcome of injecting double-bit faults



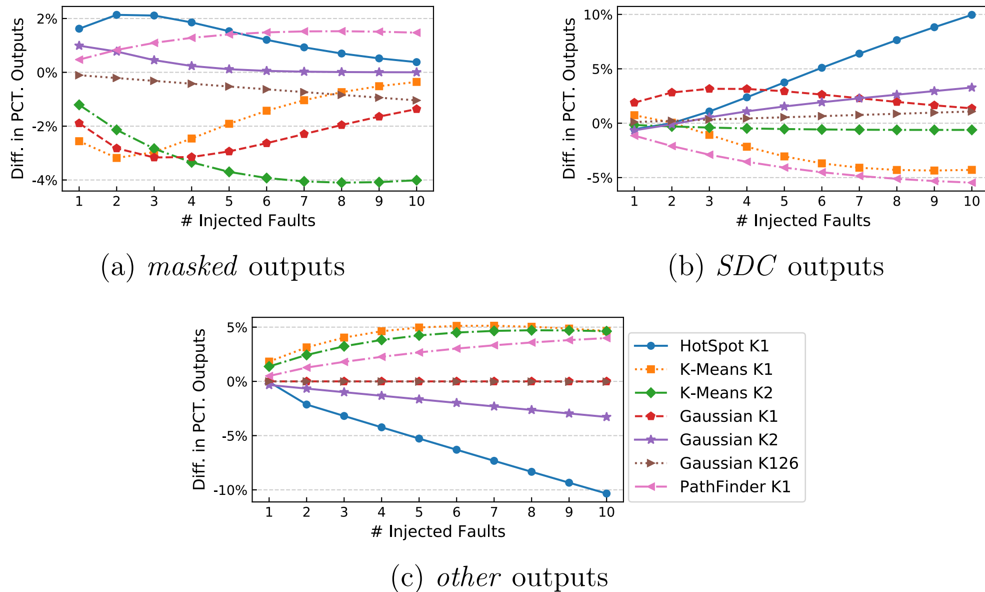
(b) Outcome of injecting triple-bit faults

Figure 7.11: Error resilience comparison of progressive fault site pruning techniques against the ground truth (baseline) for (a) injecting double-bit faults and (b) injecting triple-bit faults.

In Figure 7.11 (b), we present the the comparison of triple-bit fault injection outcomes obtained by the two techniques. We observe that the differences start to be visible as compared to single-bit fault injection (see Figure 7.9) and double-bit fault injection (see Figure 7.11(a)). The number of benchmark kernels where the discrepancy is within ±1% reduces to 9, 4, and 5 for *masked*, *SDC*, and *other* outputs, respectively. The average differences in terms of the percentage of *masked*, *SDC*, and *other* outputs increase to 1.89%, 3.45%, and 2.79%, respectively.

Notice that for some benchmark kernels (including LUD K44, LUD K45, LUD K46, 2DCONV, MVT, GEMM, and SYRK), there are already almost no *masked* outputs (i.e., *masked*% $\leq$ 5% in Figure 7.11(b)), indicating that there is no need to inject more faults

to them. The remaining kernels still have a significant portion of *masked* outputs, ranging from 6% to over 90%. For those benchmark kernels, we continue to increase the number of injected faults and show the impact of more faults on the difference between the distribution of fault injection outcomes given by the baseline and by the pruned method. Figure 7.12 shows that generally the difference increases as we inject more faults. However, the magnitude of increment varies for different type of outputs and benchmark kernels. For the majority of the benchmark kernels, the difference is always within ±5% for all three types of outputs. HotSpot is the most problematic case, the difference in terms of *SDC* and *other* outputs exceeds ±5% after injecting 6 and 5 faults, respectively. Luckily, the difference of the percentage of *masked* outputs for HotSpot is always less than 2%, which is preferable as pruning is able to provide a good estimation for this more critical type of outputs.



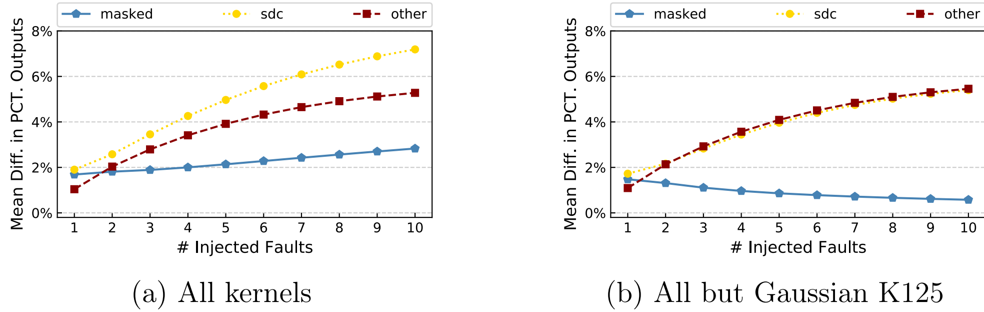
(a) *masked* outputs (b) *SDC* outputs

(c) *other* outputs

Figure 7.12: Impact of increasing number of injected faults on the difference in terms of the percentage of (a) *masked*, (b) *SDC*, and (c) *other* outputs given by the proposed pruning technique and *baseline* for selected benchmark kernels.

Figure 7.13 (a) shows an overview of the average errors by presenting the mean difference values over all kernels. As expected, the average errors increase as injecting more faults, up to 2.83%, 7.19%, and 5.28% for *masked*, *SDC*, and *other* outputs, respectively.

If Gaussian K125 is excluded and mean error values are re-calculate, the difference is then less than 2% for *masked* outputs and less than 6% for *SDC* and *other* outputs for up to 10 injected faults, see Figure 7.13 (b).



(a) All kernels (b) All but Gaussian K125

Figure 7.13: Mean values of differences in terms of percentage numbers of different fault injection outcomes calculated (a) across all kernels and (b) across all kernels but excluding Gaussian K125.

Observation 7.6 *The difference between the distribution of fault injection outcomes obtained by the proposed fault site pruning technique and baseline is acceptable, i.e., within* ±3% *for up to* 3*-bit fault injection and within* ±6% *for up to* 10*-bit fault injection.*

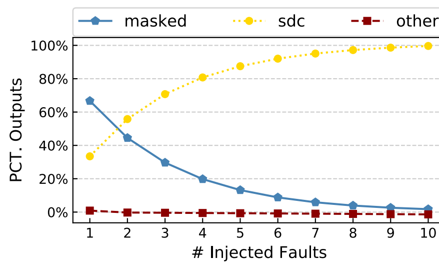
Observation 7.7 *The proposed fault site pruning gives a better estimation for masked outputs (i.e., the difference from baseline is less than* 2% *on average) than for SDC and other outputs.*

7.5.3.2 Impact of multi-bit faults

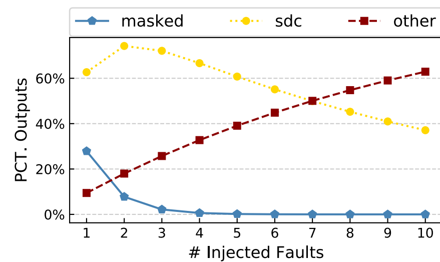
Having established in the previous section that the distribution of fault injection outcomes obtained through the proposed progressive fault site pruning technique is close to the distribution achieved by the *baseline* (i.e., the 60K experiments) under the context of multi-bit fault injection, we use the pruning outcome to explore how the error resilience characteristics of a specific application change as a function of the number of injected faults.

Figure 7.14 presents the distribution of fault injection outcomes of four representative benchmark kernels. First, we observe that the percentage of *masked* outputs reduces

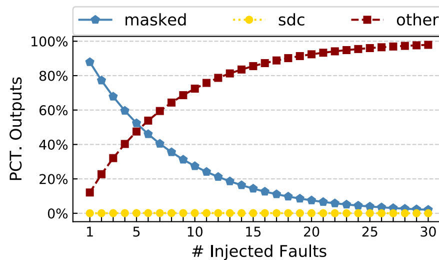
significantly as we inject more faults and ends up at 0% within 10 injected faults for most benchmark kernels, see HotSpot K1 and MVT K1 in Figure 7.14 (a) and (b), respectively. With the exception of K-Means K2 (see Figure 7.14 (e)) and PathFinder K1 (not shown as it is similar to Figure 7.14 (e)), whose percentage of *masked* outputs stabilizes at 0% for up to 30 injected faults, this shows that these two kernels are more error resilient than the others. Another exception is Gaussian K126 (see Figure 7.14 (d)), whose percentage of *masked* outputs is still over 97% with 10 faults injected, implying that this kernel is even more error resilient. In general, if the benchmark is resilient to single fault, then it is resilient to multiple faults. Looking into the code, Gaussian K2 and K126 come from the same static kernel. There is a branch condition to determine whether the program goes into a loop or if it returns. For K2, most of the times, it goes into the loop, instructions in which result in a lot of *SDC* outputs. For K126, most of the times, it takes the return branch, leading to large portion of *masked* outputs.



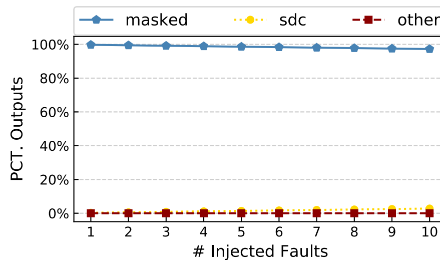
(a) HotSpot K1



(b) MVT K1



(c) K-Means K2



(d) Gaussian K126

Figure 7.14: Error resilience changes over increasing number of injected faults for representative benchmark kernels.

Observation 7.8 *For the majority of benchmark kernels, the percentage of masked outputs*

stabilizes at 0% with 10-bit fault injections.

7.6 Chapter Summary

In this chapter, we demonstrate that fault sites in GPUs are very large and hence it is impractical to inject faults at every site to gain a comprehensive understanding of the GPGPU application error resilience. To address this, we present a progressive fault site reduction methodology based on GPGPU application-specific features. The key insight behind this methodology stems from the fact that GPGPU applications spawn a lot of threads, however, many of them execute the same set of instructions. Therefore, several fault sites are redundant and can be pruned by a careful analysis of faults across threads and instructions. For additional benefits, we also considered loop iterations within the same thread and register bit positions. We pruned the associated redundant fault sites that are not necessary to capture the GPGPU application error resilience. Across a set of 10 GPGPU applications (16 kernels in total) from the Rodinia and Polybench suites, we achieve a significant reduction in the number of fault-injection experiments (up to seven orders of magnitude) needed for an accurate GPU reliability assessment. We further extend the proposed fault site pruning technique to multi-bit fault model and confirm its accuracy for multi-bit fault injection. Note that, the proposed progressive fault site pruning is application-specific and input-dependent. We leave the study of the impact of multiple inputs on pruning results for future work (see Chapter 9.1 for more discussion).

Chapter 8

A Hierarchical Approach to Enabling Low-Overhead Reliable GPU Computing

In Chapter 7, we propose a progressive fault site pruning method to systematically reduce the number of fault sites needed in an fault injection campaign. By carefully selecting pruning steps following GPGPU application specific features, we not only reduce the total number of required fault injections (in some cases to a few hundreds only while still maintaining superior accuracy), but also equivalently reduce the total time to complete the required experiments. This framework provides a good tool to deeply understand the error resilience characteristics of GPGPU applications. More specifically, in this chapter, we take advantage of this framework to study the error resilience characters of GPGPU applications not only at the kernel level, but also at finer granularities, such as CTA and warps levels. Such fine-grained understandings are beneficial to enable low-overhead yet reliable GPU computing.

In Chapter 3, we show that GPGPUs are prone to soft errors in real-world systems. Traditionally, GPGPUs are covered by various kinds of protection mechanisms such as frequent check-pointing of necessary application states, re-computation of vulnerable codes,

or other fault protection mechanisms in order to ensure reliable GPU computation [78, 74, 155, 75]. However, these methods often incur high overhead in terms of performance, power, and hardware resources [74, 155, 156, 75, 76]. To reduce these overheads, we revisit the concept of approximate computing, which acknowledges that not all faults result in unacceptable loss in application *output quality*. Therefore, if a user is willing to tolerate a *quantifiable* loss in output quality, the overhead to achieve high resilience can be avoided. We call this concept *accuracy-aware resilience*. In order to understand the interplay between approximate computing and application resilience consider, we show the effects of over 15K single-bit[1] fault-injection experiments on the output of the BlackScholes application [110] in Figure 8.1. We assume that a user can measure the acceptability of the output quality with five different thresholds ranging from very strict to very relaxed. We observe that with the default threshold (i.e., the value specified by the CUDA benchmark), a total of 89.1% faults are *benign*: 31.8% are masked (i.e., there is no change in the application output) and 57.3% can be accepted (*SDC-Accept*). Only 10.9% of the outputs are badly corrupted, either beyond user acceptability (*SDC-Reject*) or easily detectable (*DDC*), or result in crashes or hangs. If the user accepts more relaxed thresholds for the application output, the application resilience coverage, i.e., the percentage of benign faults (*masked* and *SDC-Accept*), further increases.

Motivated by the above observations, we ask the following two questions:

1. How can we systematically analyze the accuracy-aware error resilience of GPGPU applications?

2. How can we leverage this analysis to enable low-overhead reliable GPU computing?

One of the major challenges in answering these questions is to come up with a methodical approach that captures the execution flows and resource usage of thousands of concurrently executing threads in GPGPU applications. To this end, we adopt a hierarchical

[1] We focus on single-bit faults, which are the most commonly observed faults in GPGPUs [140] and are shown to be sufficient in capturing the reliability characteristics of GPGPU applications [124].

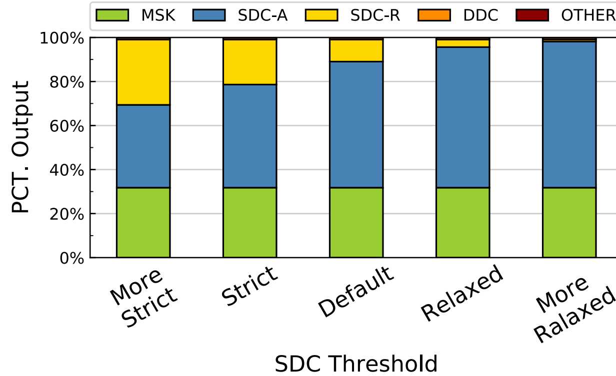


Figure 8.1: Effect of a single bit fault on the BlackScholes application output shows that a significant percentage of the fault injection runs lead to silent data corruption (SDC), which can be acceptable to a user (SDC-Accept). The percentage of SDC-Accept increases as the user-defined acceptability threshold becomes less conservative.

approach, which is inspired by the inherent GPGPU application hierarchy that arranges threads at three levels: kernels, thread-blocks (or cooperative thread arrays (CTAs) in CUDA terminology), and warps. Note that resource allocation in GPUs happens in the same order [67]. The kernel(s) are first launched on the GPUs, followed by per-core resource allocation across CTAs. The warps inside each CTA are then launched in a lock-step fashion on the single-instruction-multiple-thread (SIMT) execution lanes taking advantage of per-CTA resources. We associate this resource allocation procedure with our proposed *accuracy-aware resilience* analysis, which we show to be an effective way to determine *which resources at what levels of the application hierarchy contribute to the user acceptable (SDC-Accept) faults and hence can be protection-free.*

As a use case of the proposed hierarchical analysis of GPGPU resilience, we consider the popular re-computation model [145] as a way to provide protection and assure reliable computing: if the application outputs of the actual computation and re-computation match, then execution can be declared fault-free. Clearly, the overhead of re-computation can be severe. For example, if re-computation is performed in parallel to the actual execution, double hardware resources (e.g., core/memory/register files) would be required. If performed sequentially, the total execution time (including re-computation) would be dou-

bled. Our hierarchical approach first analyzes the error resilience of representative threads of a kernel to determine if the kernel has the level of resilience that is required by the user. If it is the case, the re-computation of the entire kernel is not required and its associated overheads are saved. On the other hand, if the resilience coverage is not adequate, we perform the error resilience analysis at a finer granularity (i.e., at CTA-level). If this analysis determines that only a fraction of CTAs do not meet the resilience coverage requirements, we require only the re-computation of such vulnerable CTAs. Consequently, the overall re-computation overhead is reduced because not all CTAs need to be re-computed. We show that our statistically-validated hierarchical approach can provide significant reduction in re-computation overhead while still meet the user requirements for application output accuracy and resilience coverage.

To the best of our knowledge, this is the first work to systematically and comprehensively analyze the *accuracy-aware resilience* for a diverse set of GPGPU applications. Specifically, we study a total of 15 benchmarks (26 kernels) and launch over 330K fault-injection runs (with an average of 10K runs per kernel), leading to the following key contributions and observations:

- We introduce the concept of *accuracy-aware resilience* to GPGPU applications, which provides more opportunities for exploring low-overhead reliable GPU computing.

- We conduct a thorough resilience analysis of a diverse set of GPGPU applications and and reach the following observations:

  a) Kernel-level: Accuracy-aware error-resilience can increase significantly if the user is able to tolerate a limited amount of inaccuracy in the application outputs.

  b) CTA-level: Accuracy-aware error-resilience can vary significantly across *groups* of CTAs. Studying a few CTAs per group is enough to represent the overall accuracy-aware error-resilience of GPGPU applications.

  c) Warp-level: Accuracy-aware error-resilience is similar across warps within a

group of CTAs. Therefore, it is sufficient to perform accuracy-aware error-resilience analysis only at the CTA-level.

- As a case study, we show that the proposed hierarchical approach can reduce the protection overheads related to re-computation based on user-defined fault tolerance and resilience coverage. Specifically, we observe that: **d)** The physical resources allocated to the entire kernel for re-computation are saved (and potentially be used for other useful work or be turned-off for power savings) if a user is able to accept a certain resilience coverage and output quality. **e)** Under stricter user-defined requirements, the re-computation overhead can still be reduced by enabling re-computation at a finer granularity (e.g., at CTA-level). Overall, the proposed hierarchical approach is able to reduce re-computation overhead while satisfying user-defined output quality and resilience coverage.

This remaining of this chapter is organized as follows. In Section 8.1, we discuss related work. Section 8.2 explains the evaluation methodology. We illustrate the design details of the hierarchical approach to thread classification in Section 8.3. The error resilience characterization is given in Section 8.4. In Section 8.5, we present a use case to show how to use the characterization results to reduce GPU resilience protection overhead. Finally, Section 8.6 gives a summary of this chapter.

8.1 Related Work

To the best of our knowledge, this is the first work that introduces the concept of *accuracy-aware resilience* in GPUs. In this section, we briefly discuss the most relevant related work.

There are a large number of studies that focus on leveraging simulation-based analysis to detect critical hardware structures that are more vulnerable to soft errors. Prior works [45, 64, 138] have conducted architectural vulnerability factor (AVF) analysis, which tracks every bit in an architecture during the application run and calculates the likelihood

of the bit that can affect the output. Although there is a large body of work on fault injection models/frameworks [92, 42, 88, 89, 113, 117, 123, 128, 126, 23, 95, 49, 50, 51] in the context of CPUs, only a limited set of fault injector models have been proposed for GPUs [154, 59, 44, 85]. For example, Yim et al. [154] build a source-to-source translator, SWIFI, to investigate error resilience in GPUs and demonstrate that the ratio of silent data corruption (SDC) in GPUs is much higher than that observed in CPUs.

GPU-Qin [44] uses the number of dynamic instructions (DI) per thread as a proxy for thread behavior. The rationale is that threads with the same dynamic instruction count are likely to execute the exact same set of instructions, thus resulting in similar error resilience behavior. GPU-Qin groups threads with the same dynamic instruction counts at the kernel-level and randomly chooses a single thread per group for fault injection. There are two primary differences between our work and GPU-Qin. GPU-Qin groups threads strictly based on the DI count. Therefore, threads with different DI count are grouped in different groups. Our hierarchical based method may put threads with the different DI count in the same group based on additional hierarchical information (Section 8.4.2). This allows higher overhead reduction compared to GPU-Qin because vulnerable threads are encapsulated into fewer CTAs, which then can be recomputed. On the other hand, a DI-count based method can find vulnerable threads spanning across a large number of CTAs (because threads with different DIs are present in large number of CTAs), leading to larger overhead.

While the purpose of fault protection is to completely avoid faults, approximate computing instead explores the trade-off between accuracy, performance, and energy efficiency. Prior studies have considered this trade-off in specific areas including bioinformatics [96, 63], performance analysis [143], data mining [97], and image recognition [94]. Approxilyzer [145] has been proposed to evaluate the three-way trade-off among output quality, resilience coverage, and overhead reduction. However, it is built for single-threaded CPU applications only and is not clear how it can be extended for parallel GPGPU applications with thousands of threads and billions of fault sites.

8.2 Evaluation Methodology

In this section, we discuss the benchmarks and corresponding evaluation metrics used in this chapter followed by an overview of the experimental framework.

8.2.1 Benchmarks and Evaluation Metrics

We select benchmarks that cover various workloads from diverse areas, such as image processing, finance, linear algebra, physics, molecular dynamics, and data mining. We select 15 GPGPU applications from widely-used benchmark suites, including CUDA [110], SHOC [37], Rodinia [31], and Mars [60], see Table 8.1. As kernels of GPGPU applications normally implement independent modules, we perform resilience coverage analysis separately for each kernel. For benchmarks with more than one kernel, we randomly select at most four kernels for fault injection experiments. In the rest of this chapter, if the kernel index is not specified, it implies that the benchmark only contains one kernel.

In order to determine whether a certain SDC output is acceptable or not, we need a metric and threshold value to quantitatively measure the difference between the outputs of fault-injected and fault-free runs. Choosing the most appropriate metric and threshold requires domain knowledge. *We anticipate that the evaluation metrics/thresholds are provided by the user or system administrator.* In addition, we also provide several choices of commonly used metrics and threshold default values [145]. For most applications, we choose widely-used metrics such as mean squared error (MSE)[2] and percentage loss (PercLoss)[3]. For certain applications, we use domain-specified metrics. For example, RAY and JPEG, which are image processing applications, are evaluated by the MSE of images pixel by pixel. For Neural Network (NN), the prediction accuracy for the faul-free run is 100%, we use the difference to this value as the evaluation metric. For Sort, the result of which is a ranked list, we use the commonly used Ranked Biased Overlap (RBO) [148] to

[2] $MSE = \frac{1}{n}\sum_{i=1}^{n}(X_i - Y_i)^2$, where X and Y are two vectors of size n.

[3] $PercLoss = \frac{\#\ miss_match}{\#\ total} \times 100\%$, where $\#\ miss_match$ is the number of different values in the fault-free and fault-injected outputs, and $\#\ total$ is the total count of values.

Table 8.1: List of Applications with Evaluation Metrics and Thresholds.

Suite	Benchmark	Evaluation Metric	Default Threshold
CUDA	BlackScholes (BS)	L1 norm*	$1e-6$
	Ray Tracing (RAY)	MSE of images	0.1
	Convolution Separable (CONS)	Relative L2 norm*	$1e-6$
	Fast Walsh Transform (FWT)	L2 norm*	$1e-6$
	JPEG	MSE of images	0.1
	Kmeans (KMN)	MSE of centroid coordinates	0.01
	Laplace3D (LPS)	RMS error*	$1e-6$
	Neural Network (NN)	Difference in prediction accuracy	1%
	Scalar Product (SCP)	L1 norm*	$1e-6$
	Scan Large Array (SLA)	PercLoss	1%
Mars	WordCount (WC)	Difference in word count	1
Rodinia	HotSpot (HS)	MSE of output temperature list	0.01
SHOC	Breadth-First Search (BFS)	PercLoss	1%
	Molecular Dynamics (MD)	PercLoss	1%
	Sort	Ranked Biased Overlap	0.01

* indicates metrics and thresholds as provided by benchmarks. Alternatively, the threshold values can be provided by the user or administrator.

quantify the difference between fault-free and fault-injected outputs. Table 8.1 shows the evaluation metrics and default threshold values for every benchmark. Besides the default threshold, we also evaluate application output with two more strict and two more relaxed threshold values, yielding to a total of five levels of SDC threshold values (as shown in Figure 8.1).

8.2.2 Evaluation Framework

Figure 8.2 gives an overview of the fault injection and evaluation framework, which consists of four components. For a given benchmark kernel, we first use the ❶ **Classifier**, to identify a number of threads from the massive number of parallel threads in the kernel.

Then, for every candidate thread, we resort to the ❷ **Fault Injector** to determine potential fault-injection spots, such as instructions and bit positions in destination registers. Next, in ❸ **Simulation-Level Analyzer**, for every selected spot, we use the GPGPU-Sim simulator to execute the kernel multiple times. We inject only one fault per run. Finally, the output of fault-injected runs are passed to ❹ **Quality Analyzer** to investigate the accuracy-aware resilience for the benchmark kernel. In the rest of this section, we discuss the design details of these four components.

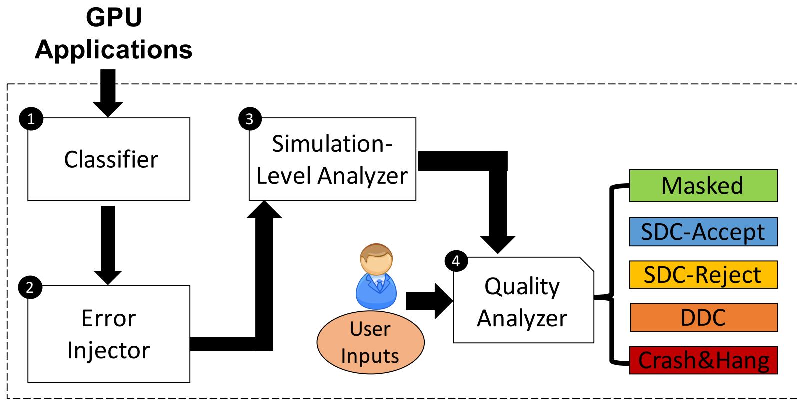


Figure 8.2: A high-level view of fault injection and evaluation framework.

❶ **Classifier.** GPGPU applications can contain a massive number of threads. Therefore, it is unrealistic to perform fault-injection runs on every thread. Consequently, we have to identify a fraction of representative threads, which is a challenging open problem. We realize this with a hierarchical (i.e., CTA-level and warp-level) classification and thread selection method. In Section 8.3, we illustrate the motivation and effectiveness of this approach.

❷ **Fault Injector.** The goal is to determine interesting and meaningful *fault sites*. We used the popular single-bit fault injection model [44, 59, 104] to evaluate the effect of soft errors in GPUs. These faults affect the functional units such as arithmetic-logic units (ALUs) and the load-store units (LSUs), which are *not* protected in commercial GPUs. A fault site contains three aspects of data: (1) *tid* indicates the candidate thread, (2) *inst_id* and *sim_cycle* identify the instruction and the simulation cycle it is executed (*sim_cycle*

is necessary because the same instruction can be executed many times, i.e., if inside a loop), (3) *bit_pos* tells which bit to flip in the register.

The first element *tid* is determined in and passed by ❶. Next, we profile the thread with the GPGPU-Sim simulator to collect instruction-related execution details, including the instruction type, the simulation cycle it is executed, and the destination register type. There can be tens to thousands of dynamic instructions in one thread. In order to control the number of fault sites, we randomly sample a few iterations for instructions inside *loop* blocks. We also select all instructions outside *loop* blocks to make sure we cover all types of instructions. Finally, for every selected instruction, we flip one bit in its destination register. The *bit_pos* is chosen from a set of pre-selected bit position candidates that are evenly spread in the register. Those bit positions are selected to cover a range of positions in registers, as it is impractical to conduct experiments on every single bit.

❸ **Simulation-Level Analyzer.** This component accepts the selected list of fault sites and performs fault-injection experiments using GPGPU-Sim [24], a widely-used cycle-accurate GPU architectural simulator. We choose GPGPU-Sim over hardware-level tools (e.g., SASSIFI [59]) because of its capability of capturing all the detailed micro-architectural and architectural states of all hardware components (e.g., registers, cache lines). Its functional model is validated against real hardware [24], so that the simulation results are representative. Note that, we only inject a single fault per application run and record the detailed execution information provided by GPGPU-Sim, including the fault site, the original and impacted values in the register, and the final output. For all chosen benchmarks (see Table 8.1), we launch 330K fault-injected runs, with an average of 10K runs per benchmark kernel. In Section 8.4.4 we show that the data provide an ample state space for results of statistical significance.

❹ **Quality Analyzer.** In this component, we investigate the fault-injected outputs and evaluate *accuracy-aware resilience*. Soft errors can have diverse effects on the execution of an application. For example, sometimes faults may not lead to any difference between the fault-injected and the fault-free outputs, thus are classified as *Masked* faults. In some

cases, although faults allow the application to execute completely, the application output is incorrect. Such faults are typically classified as *Silent Data Corruption* (SDC) faults. In certain circumstances, users can accept the approximate version of the application output. Therefore, we further classify SDC faults into *SDC-Accept* and *SDC-Reject* according to the user acceptability threshold. Furthermore, some corrupted results can be easily detected (i.e., irregular negative value, infinite or NaN value), we classify these faults as *Detectable Data Corruption* (DDC). Lastly, faults can also result in crashes or hangs. In summary, as shown in Figure 8.2, we classify the output of fault-injected runs into five categories: (1) *Masked*, (2) *SDC-Accept*, (3) *SDC-Reject*, (4) *DDC*, and (5) *OTHER*. The first two faults are *benign* and define the resilience coverage of the application, while the rest are *non-benign*.

8.3 A Hierarchical Approach to Thread Classification

In the previous section, we discuss the design details of the components in the fault injection and evaluation framework, except for the thread classification component (❶) in Figure 8.2. In this section, we explain in detail how we systematically perform the hierarchical classification and thread selection.

Fault-injection models are commonly used to explore resilience characteristics of CPUs and GPUs. Typically, in the CPU domain, faults are uniformly injected throughout the entire application execution. The effect of each fault is examined in isolation via separate application runs. Unfortunately, such an approach is tedious for GPUs, as there can be hundreds to thousands of threads running concurrently. Injecting faults to every thread dramatically increases the model complexity and results in an astronomical number of fault-injection runs. To address this issue, we have to select a manageable number of representative threads for fault injection.

8.3.1 Multi-level Classification and Thread Selection

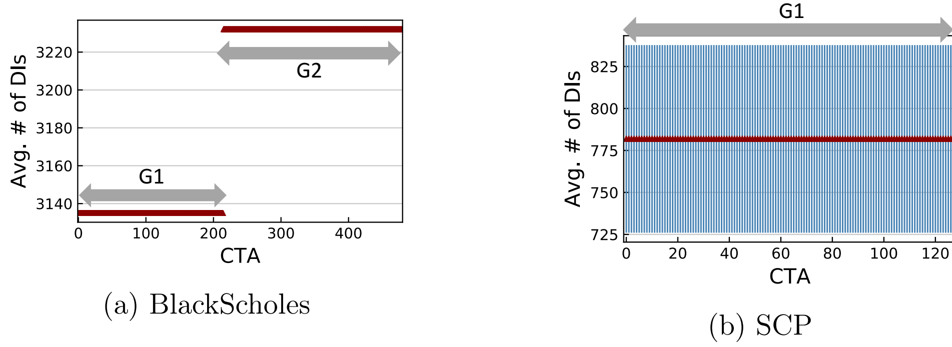
We propose and evaluate a hierarchical approach for thread grouping. Following the hierarchy of GPGPU applications, we classify threads at the CTA and warp levels. We group CTAs (or warps) based on the distribution of thread dynamic instruction (DI) counts[4], which has been shown to be an effective proxy for accurately capturing the error resilience of threads [44, 104]. The rationale is that threads with the same DI count are likely to execute the exact same set of instructions, thus resulting in similar error resilience behavior. Such rationale is also confirmed with millions of fault injection experiments [104]. We consider the mean and one standard deviation of DI counts to quantitatively compare different CTAs (or warps). Then, from each CTA (or warp) group, we randomly select a limited number of threads for fault injection.

8.3.1.1 CTA-level classification

Regular CTA Analysis. First, we illustrate the distribution of DIs at the CTA level for different benchmarks. Figure 8.3 focuses on two regular benchmarks: BlackScholes and SCP. The x-axis indicates the index of CTAs, while the red triangle and the blue error bar correspond to mean and one standard deviation of DI counts, respectively. CTAs are sorted by the average DI counts in the ascending order along the x-axis. Since we classify CTAs with similar DI distribution as one group, there exist two distinct CTA groups for BlackScholes (see Figure 8.3(a)). In addition, each group only contains one type of thread (i.e., the standard deviation of DI counts is 0). For such regular benchmarks, we only group at the CTA level, since grouping at the warp level makes no difference. Figure 8.3(b) reports on SCP, another regular benchmark. All CTAs share the same mean and standard deviation of DI counts while the standard deviation is higher than that in Figure 8.3(a). For kernel-level and CTA-level analysis, we classify all CTAs in SCP as a

[4]Detailed kernel/CTA/warp/thread information can be obtained through the GPGPU-Sim simulator [24].

single group. Because of the high variance in DI counts in certain CTAs, we also perform warp-level analysis (see Section 8.3.1.2).

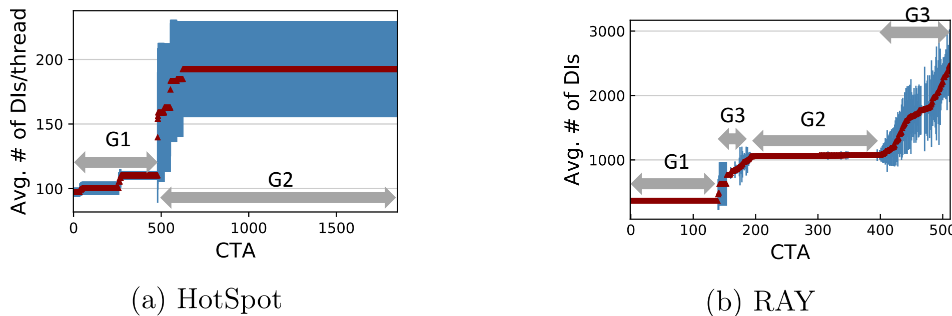


(a) BlackScholes

(b) SCP

Figure 8.3: Distribution of thread dynamic instruction (DI) counts at the CTA level for regular benchmarks (a) BlackScholes and (b) SCP. **The red triangle indicates the average and the blue error bar indicates one std.**

Irregular CTA Analysis. Figure 8.4 illustrates two irregular benchmarks: HotSpot and RAY, which exhibit divergence in the DI distributions due to branch instructions in their kernels. We classify CTAs in HotSpot into two groups (see Figure 8.3(a)): group *G1* (regular group) contains CTAs with low standard deviation of DI counts while group *G2* (irregular group) contains CTAs with diverse threads. Likewise, in RAY (see Figure 8.4(b)), CTAs with no variance in DI counts are grouped into regular groups *G1* and *G2*, while all other CTAs are classified into the irregular group *G3*.



(a) HotSpot

(b) RAY

Figure 8.4: Distribution of thread dynamic instruction (DI) counts at the CTA level for irregular benchmarks (a) HotSpot and (b) RAY. **The red triangle indicates the average and the blue error bar indicates one std.**

Effect of Input. We explore the question: does the CTA grouping method change with

application input? If not, this implies that it is possible to profile the kernel once and the resulted grouping is applicable to other inputs. To explore this, we feed HotSpot and RAY with three inputs: Small, Medium, and Large. Table 8.2 shows the effect of various inputs on group "popularity" (i.e., the percentage of CTAs in that group).

Table 8.2: The Impact of Different Inputs on CTA Group Popularity for HotSpot and RAY. **Notation: GRP-S/M/L=the percentage of CTAs in that group with Small/Medium/Large input, R=regular, IR=irregular.**

Benchmark	Grp.	Type	GRP-S	GRP-M	GRP-L
HotSpot	*G1*	R	31%	26%	26%
	G2	IR	69%	74%	74%
RAY	*G1*	R	25%	25%	27%
	G2	R	9%	22%	28%
	G3	IR	66%	53%	45%

* R: regular group; IR: irregular group.

We observe that for both benchmarks, the number of CTA groups, as well as their types (regular or irregular), is the same in all three inputs, implying that the CTA grouping strategy is input-independent. Additionally, we notice that group popularity changes with different inputs. For example, for HotSpot, the popularity of *G1* starts from 31% for *Small* input, then decreases but stabilizes at 26% as the input size increases. For RAY, the popularity of *G1* is quite stable but that of *G2* increases significantly from 9% to 28% with larger input sizes. We also explore the impact of input size on CTA grouping strategy for other benchmarks and this observation persists. For brevity, we do not show those results. In sum, the number of groups persists across different inputs while their popularity may change.

8.3.1.2 Warp-level classification

As the next level of the GPU resource allocation procedure, we focus on the warp level to explore whether heterogeneity in terms of dynamic instruction counts exists within CTAs. Figure 8.5 shows the mean and one standard deviation of DI counts at the warp level for

SCP, which is different from the CTA-level (compare to Figure 8.3(b)). At the warp level, we are able to classify warps into four groups: regular groups *G1*, *G2*, and *G3* with no variance in DI counts, and irregular group *G4*.

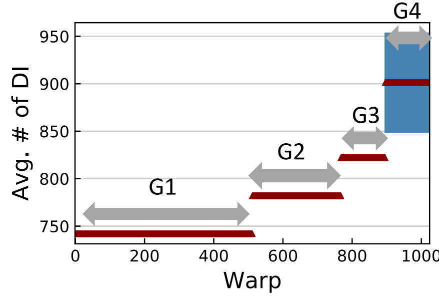
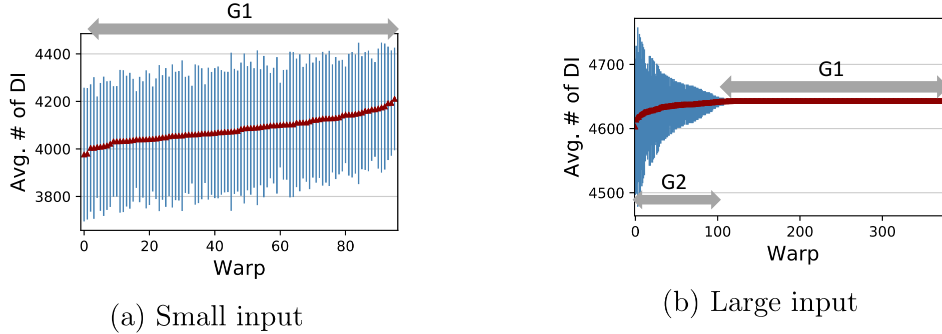


Figure 8.5: Distribution of thread dynamic instruction counts at the warp level for SCP. **The red triangle indicates the average and the blue error bar indicates one std.**

We also investigate whether warp-level grouping is input-independent and this holds for all benchmark kernels except MD. Figure 8.6(a) and (b) show the warp-level plots for MD k1 using Small and Large inputs, respectively. For Small input, we classify all warps into one irregular group while for Large input, we classify warps into two groups: the regular *G1* and the irregular *G2*. However, if we further explore the warp-level DI counts in MD k1 with Large input, we find that all warps look very similar. In fact, almost all (i.e, $\geq$ 94%) threads in most (i.e., $\geq$ 98%) warps in *G2* share the same DI count as threads in *G1*. That is, only 1 or 2 threads out of all 32 threads per warp are different. Therefore, all warps in MD k1 with Large input can also be classified as one group, just as in the classification for MD k1 with Small input. Consequently, even though different inputs may change the look of error bars, it does not truly impact the resulting warp grouping strategy. In other words, we can still apply the same warp-level grouping method derived from one input to others.

8.3.1.3 Classification result and thread selection

We apply the CTA-level and warp-level grouping method described above to every benchmark kernel. Table 8.3 shows the classification results. Column *Grp. Level* indicates the



(a) Small input

(b) Large input

Figure 8.6: Distribution of thread dynamic instruction counts at the warp level for MD k1, using two inputs: (a) Small and (b) Large. **The red triangle indicates the average and the blue error bar indicates one std.**

classification level, i.e., CTA or warp. Recall that we only consider the warp level for SCP-like benchmark kernels. Column *# DI Grp.* shows the number of groups classified by the distribution of DIs in CTA or warp, while Column *% R-Grp* points out the percentage of regular groups (i.e., groups with low to no variance in DI counts). Naturally, due to the simplicity in thread selection, regular groups are preferable. Fortunately, we observe a significant percentage of regular groups in most benchmark kernels, varying from 26% to 100% with an average of 82%. We also explore the error resilience group-wise and further combine groups that share similar resilience characteristics. Column *# ErrDist Grp.* indicates such refined group counts. More details are discussed in Section 8.4.

Having determined the grouping level and strategy, the next step is to select a limited number of threads per group for fault injection runs. For regular groups, where all threads share similar dynamic instruction counts, it is straightforward to randomly select one thread per group. For irregular groups, which contain a variety of different threads, we randomly select a limited number of threads based on the frequency of their DI counts.

Observation 8.1 *Only a few groups of CTAs are different in terms of the number of dynamic instructions they execute.*

Observation 8.2 *Only a few warps within the selected groups of CTAs are different in terms of the number of dynamic instructions they execute.*

Table 8.3: CTA-level and Warp-level Classification for Benchmark Kernels. **Notation: *%R-Grp.*= % regular groups over all groups, # *DI Grp.*=# of groups classified by dynamic instruction counts, # *ErrDist Grp.*=# of groups refined by fault distribution.**

Benchmark	**# CTA**	**# Warp**	**Grp. Level**	**% R-Grp**	**# DI Grp.**	**# ErrDist Grp.**
BlackScholes	480	1920	CTA	100%	2	1
RAY	512	2048	CTA	55%	3	3
CONS k6	1152	4608	CTA	100%	3	1
CONS k7	2304	4608	CTA	100%	2	1
FWT k6	1024	16384	CTA	100%	1	1
FWT k13	128	1024	CTA	100%	1	1
JPEG	512	1024	CTA	100%	1	1
KMN k1	121	968	CTA	100%	2	2
KMN k2	121	968	CTA	100%	2	2
LPS	128	512	CTA	0%	3	1
NN k4	1000	1000	CTA	100%	2	2
HotSpot	1849	14792	CTA	26%	2	2
BFS k3	20	320	CTA	90%	3	3
BFS k9	20	320	CTA	90%	3	3
BFS k11	20	320	CTA	85%	3	3
WC k114	512	2048	CTA	94%	2	2
WC k5	1	8	Warp	75%	2	2
WC k91	32	256	Warp	85%	5	2
SCP	128	1024	Warp	87.5%	4	1
SLA k256	8	64	Warp	87.5%	4	1
SLA k258	8	64	Warp	0%	1	1
MD k1	48	384	Warp	100%	1	1
MD k3	48	384	Warp	100%	1	1
Sort k8	512	4096	Warp	88%	2	1
Sort k20	512	4096	Warp	87.5%	2	1
Sort k24	512	4096	Warp	87.5%	2	2

Observation 8.3 *Hierarchical grouping is not sensitive to the type or size of the input.*

8.4 Hierarchical Approach to Error Resilience Characterization

Having demonstrated the rationale and methodology for the hierarchical thread classification and selection method, in this section we characterize and analyze benchmark resilience.

8.4.1 Application Kernel Level Characteristics

We start the resilience coverage analysis at the highest level, that is investigating the benchmark kernel as a whole. We evaluate the soft-error resilience by computing the distribution of fault injection outcomes, which is the percentage of each type of fault (i.e., *Masked, SDC-Accept, SDC-Reject, DDC, and Others*) among all fault-injection runs. Recall that we launch over 330K fault-injection runs, with an average of 10K runs per kernel. In Section 8.4.4, we validate statistically that 10K runs are sufficient to obtain the error resilience profile of GPGPU applications.

8.4.1.1 Scope of accuracy-aware resilience

Figure 8.7 presents the distribution of fault injection outcomes evaluated with the default SDC threshold of every benchmark kernel listed in Table 8.1. Every stacked bar represents the fault distribution of one benchmark kernel. The first impression is that for all benchmarks, the majority of soft-errors are masked, i.e., they are imperceptible to the end user. The actual percentage numbers of *Masked* faults vary from 31.8% in BlackScholes to 100% in Sort k20, SLA k258, and CONS k7. For CONS k6, there are very few non-masked faults (i.e., $\leq 1\%$), which are barely visible in Figure 8.7. We check the number of loop iterations in those benchmarks with close to 100% *Masked* outputs and the low number (≤ 6) confirms that the results are not biased due to sampling. Such a large portion of *Masked* faults implies that the protection effort for these runs is perhaps not necessary.

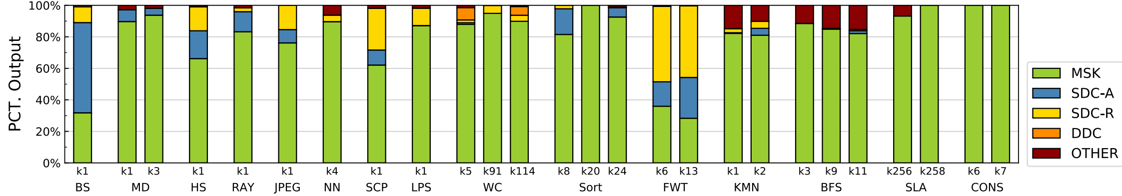


Figure 8.7: Distribution of fault injection outcomes at benchmark kernel level. (SDC faults are evaluated with the default threshold values.)

Secondly, we notice that the majority of the benchmark kernels present a non-negligible percentage of *SDC* faults. In previous works, these faults are deemed unacceptable. In approximate computing users may be willing to trade corrupted output with lower resilience overhead and better performance, as long as the "degree of corruption" is within expected ranges. For this reason, we further divide the *SDC* results into *SDC-Accept* and *SDC-Reject*. We observe that those benchmark kernels with a large portion of *SDC* faults also exhibit a significant percentage of *SDC-Accept* faults. Note that the fault distribution in Figure 8.7 is evaluated with the application default threshold values. The percentage of *SDC-Accept* is expected to increase when the benchmark is evaluated with relaxed threshold values (see also Figure 8.1). From Figure 8.7, we see that the percentage of *SDC-Accept* faults can be very high in some benchmarks, such as 12.8% in RAY, 17.7% in HotSpot, 15.5% in FWT k6, 25.9% in FWT k13, and even 57.3% in BlackScholes. While in some other benchmarks (i.e., LPS, SLA, and CONS), there are little to no *SDC-Accept* faults. Note also that these benchmarks have a significant percentage ($\geq$ 89%) of *Masked* faults.

From the domain perspective, image processing applications such as RAY and JPEG are resilient to soft-errors, as minor changes in output images are barely distinguishable by the end users. NN also digests single bit flips well. Those soft errors slightly impact the weights of trained neural networks, thus barely result in wrong outputs. In contrast, SCP and FWT are more sensitive to soft errors, see the percentage of *benign* faults (*Masked* and *SDC-Accept*) in Figure 8.7.

8.4.1.2 Sensitivity to input size

Besides the kernel-level investigation on accuracy-aware resilience for GPGPU applications, it is also interesting to understand the impact of different inputs on fault distribution. Toward this, we apply two choices of inputs, i.e., Small vs. Large, on five of the benchmark kernels, see Figure 8.8.

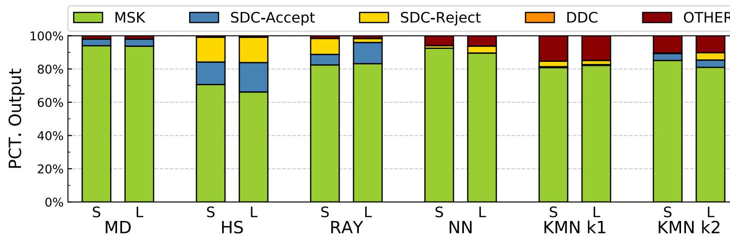


Figure 8.8: Impact of Small and Large inputs on fault distribution.

For NN and KMN k1, whose scope of *SDC-Accept* is negligible (i.e., $\leq 1\%$), we observe high similarity in the fault distribution when using different input sizes. For other benchmark kernels, using large input leads to a decrease in the percentage of *Masked* faults, specifically to 4.1% for KMN k2, 0.3% for MD, and 4.5% for HotSpot. Fortunately, for these kernels, the percentage of *SDC-Accept* increases correspondingly with large input, resulting in a similar scope of *benign* faults comparing to the small case. KMN k2 is the only exception, where the increase of *SDC-Accept* is less than the increase of *SDC-Reject*. Moreover, for RAY, whose scope of *Masked* faults is not impacted by the input size, using the large input makes the kernel more error resilient by having a larger percentage (i.e., 6.6% more) of *SDC-Accept* faults.

Observation 8.4 *There is an ample scope of SDC-Accept faults in some GPGPU applications.*

Observation 8.5 *Using large input typically preserves or increases the scope of resilience coverage, i.e., benign outputs.*

8.4.2 CTA Level Characteristics

Consistent with the hierarchical classification at the CTA level (see Section 8.3), we perform fault injection runs for every CTA group, in order to explore whether fault distributions vary across different CTA groups.

In Table 8.3, we show the number of CTA groups based on the distribution of dynamic instructions in CTAs of the benchmark kernel (see Column "*# DI Grp*"). We further combine groups that share similar fault distribution and the final number of groups is shown in Column "*# ErrDist Grp.*". Clearly, *# ErrDist Grp.* $\leq$ *# DI Grp*. We observe that for BlackScholes, CONS, and LPS, all *DI* groups are combined into one *ErrDist* group due to the similarity in the group fault distribution. The reason is that for BlackScholes, though there are two *DI* groups (both are regular ones), their average numbers of dynamic instructions are very close (3135 vs. 3232), yielding to similar fault resilience characteristics. Similar reason applies to CONS. For LPS, all its three *DI* groups are irregular, and although they have different average number of dynamic instructions, the major composite threads are the same (i.e., have the same dynamic instruction counts), resulting in similar resilience characteristics.

Except for the three aforementioned benchmarks, the rest of benchmark kernels share different fault distribution at the CTA level. Figure 8.9 shows the stacked bar plots for the fault distribution of every *ErrDist* group for 10 benchmark kernels (others are not shown due to limited space). We observe that the fault distribution can be significantly different among *ErrDist* groups. First, the composition of fault distribution can be different. In KMN k2, almost all soft errors are masked in *G1* while there is large portion of *SDC-Accept*, *SDC-Reject*, and *Other* faults (i.e., 4.4%, 4.4%, and 10.2%, respectively) in *G2*. Such observation also exists for KMN k1 and WC k114. Second, for some other kernels, certain *ErrDist* groups can have more percentage of *SDC-Accept* faults, including RAY *G2* and *G3* (14.1% and 15.9%, respectively), HotSpot *G2* (23.4%), and BFS k11 *G1* (3.0%). Furthermore, NN k4 *G1* and BFS k11 *G3* present a notable larger percentage of *SDC-*

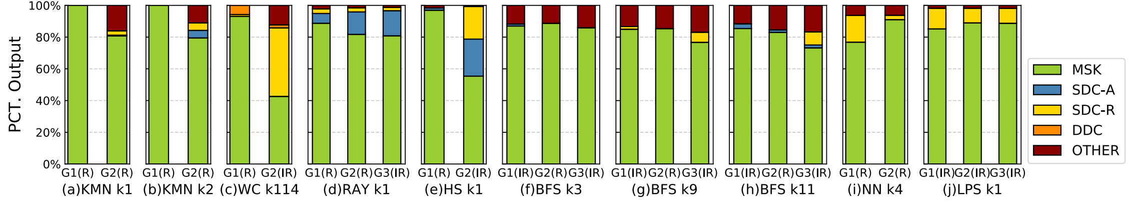


Figure 8.9: Error resilience characteristics at CTA level. **Each bar is distinguished by its group name and whether it is regular (R) or irregular (IR).**

Reject faults (i.e., 16.9% and 8.2%, respectively), which have the potential to be converted to acceptable output with relaxed threshold values. In contrast, the difference between the fault distribution of *ErrDist* groups in some benchmarks (i.e., BFS k3) can be small, but the percentage of *benign* faults in their *ErrDist* groups is high ($\geq$ 85%).

In general, we observe that regular groups tend to have a larger portion of *benign* faults than irregular ones. Furthermore, if we only focus on *Masked* faults, which are by definition always *benign*, the regular groups always have a large portion as compared to irregular ones. For instance, the percentage of *Masked* faults in WordCount k114 is 93.1% and 42.6% in regular *G1* and irregular *G2*, respectively. In HotSpot, the percentages are 96.9% in the regular group and 55.4% in the irregular one.

Observation 8.6 *A significant percentage of CTA groups are more resilient (i.e., have high percentage of SDC-Accept outputs) than other groups.*

8.4.3 Warp Level Characteristics

Previously, we show that CTA groups have distinct fault distribution, especially comparing the regular groups with the irregular ones. Here, we are interested in whether such heterogeneity persists in the warp level. We use the warp-level grouping strategy (described in Section 8.3) to classify warps within the same CTA groups. Figure 8.10 (a) to (d) show the fault distribution of every warp-level *DI* group for SCP, Sort k24, WordCount k5, and WordCount k91, respectively. For SCP (see Figure 8.10(a)), all four DI groups share similar percentage of faults, while only G1 contains slightly more SDC-Accept outputs than

the others. Such similarity in the fault distribution among warp groups is also observed for Sort k24 in Figure 8.10(b). Consequently, for both kernels, there is only one *ErrDist* group.

On the other hand, we observe significant difference in error-resilience among warp groups in WordCount k5 and k91 (see Figure 8.10 (c) and (d), respectively). Some groups (i.e., G1 for WordCount k5 and G1 and G2 for WordCount k91) are more resilient to injected faults than other groups. Recall that the results shown in Figure 8.10 are evaluated with default threshold values. By varying levels of thresholds (results are not presented here due to lack of space), we observe that warp groups exhibit different sensitivity to the threshold values. For SCP, all the warp groups increase the percentage of SDC-Accept almost at the same amount as we relax the SDC threshold values. In contrast, for WordCount k91, G5 is more sensitive to relaxed threshold values (i.e., we observe an increase increase in the percentage of SDC-Accept) than the other warp groups.

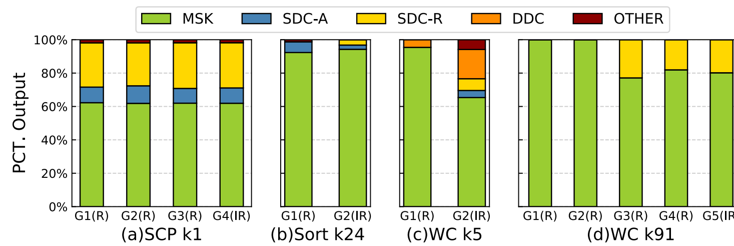


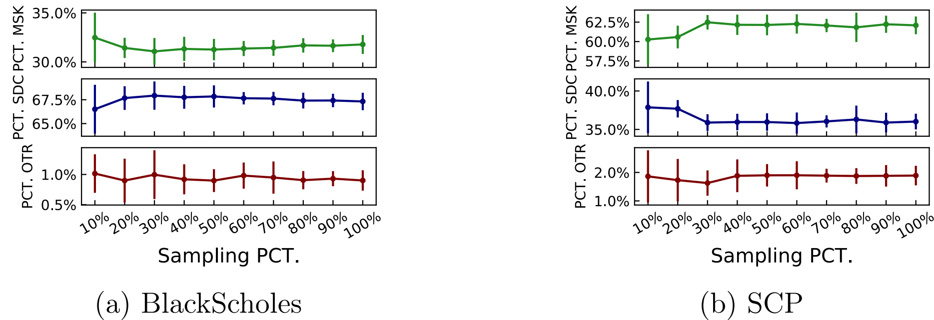
Figure 8.10: Error resilience characteristics at warp level. **Each bar is distinguished by its group name and whether it is regular(R) or irregular(IR).**

Observation 8.7 *Similar to CTA-level analysis, some warps are more resilient than others.*

8.4.4 Statistical Validation

We have shown the resilience characteristics for benchmarks at the kernel, CTA, and warp levels. The vast parallelization of the GPGPU applications makes the generation of all possible fault sites not possible. To evaluate the statistical significance of our result, for every

benchmark kernel, we randomly sample 10% of the entire space of generated fault sites and gradually add 10% until we reach 100% (i.e., all generated sites). For every increment, we calculate the 95% confidence interval. Figure 8.11 reveals how the percentage of fault changes over increasing sample sizes for BlackScholes and SCP, respectively. It is clear that the fault percentage fluctuates significantly in the initial increments, indicating that the sample space in insufficient to reach results of statistical significance, but becomes steady after the sampling percentage exceeds 80%. Moreover, we see significant overlaps across the confidence intervals, which suggests that our experiments do capture the "unknown" means of the fault distributions. In fact, we observe that the average error margin is 1.27%, 0.75%, and 0.75% for the percentage of *Masked*, *SDC*, and other faults (including *DDC*, crashes, and hangs), respectively (see the ranges of the y-axes of the graphs in Figure 8.11). Note that we perform such analysis for every benchmark kernel but cannot report results here due to lack of space. These results show that the number of experiments is sufficient to obtain average faults of statistical significance.



(a) BlackScholes (b) SCP

Figure 8.11: Changes in the percentage of faults with increasing sample size for (a) BlackScholes and (b) SCP. *PCT.MSK*, *PCT.SDC*, and *PCT. OTR* indicate the percentage of masked, SDC, and other (including DDC, crashed, and hangs) faults, respectively. **Error bars give the 95% confidence intervals.**

8.5 Use Case: Reducing Protection Overhead

In this section, we leverage on the various observations of our characterization study to improve on application resilience while maintaining reduced overhead. We first discuss the

trade-off among the following metrics.

1. **Resilience Coverage (RC) and Output Quality (OQ)**: The *perfect output quality* refers to only accepting *Masked* outputs. However, as shown in Section 8.4, there exists a large scope of *SDC-Accept* outputs of GPGPU applications. These tolerable outputs provide the opportunity of improving the Resilience Coverage (RC), which is defined as the percentage of runs with *benign* faults (i.e., *Masked* and *SDC-Accept*). Acceptable resilience coverage is application and user dependent [145].

2. **Overhead Reduction (OR)**: To improve GPGPU application resilience, we consider a re-computation model that computes the kernel again and compares its output with the actual execution output for any anomalies. As a baseline, we assume all CTAs of the kernel are vulnerable (i.e., do not meet the Resilience Coverage requirement) and hence need to be re-computed at the expense of additional physical resources. In the worst case, these resources are twice of the total resource required for the actual computation.

We focus on how our accuracy-aware resilience characterization can help in reducing the physical resource requirements. For example, if our characterization shows that 50% of CTAs are not vulnerable, then only 50% additional physical resources are required for re-computation. In the remaining section, we consider two different output quality (OQ) thresholds:

1. *Perfect OQ*: includes *Masked* outputs only.

2. *Default OQ*: includes *Masked* outputs and *SDC-Accept* outputs (evaluated with default thresholds, see Table 8.1).

Table 8.4 shows the trade-off between resilience coverage and re-computation overhead for different benchmark kernels. Under "Kernel-Level", the "Perfect OQ (OR)" column provides resilience coverage and (protection overhead reduction) that considers Perfect OQ

while the “Default OQ (OR)" column provides resilience coverage and (protection overhead reduction) that considers Default OQ. For some benchmark kernels, we can further gain on overhead reduction by considering thread groups at a finer granularity (see column “Default OQ (OR)" under “CTA-/Warp-Level".)

Table 8.4: Resilience Coverage vs. Overhead Reduction.

Benchmark	**Kernel-Level**		**CTA-/Warp-Level**
	Perfect OQ (OR)	**Default OQ (OR)**	**Default OQ (OR)**
BlackScholes	**31.8% (0%)**	89.0% (100%)	–
RAY	**83.2% (0%)**	96.0% (100%)	–
Sort k8	**81.5% (0%)**	97.8% (100%)	–
JPEG	**76.1% (0%)**	**84.6% (0%)**	–
SCP	**62.1% (0%)**	**71.6% (0%)**	–
FWT k6	**36.0% (0%)**	**51.5% (0%)**	–
FWT k13	**28.3% (0%)**	**54.2% (0%)**	–
HotSpot	**66.2% (0%)**	**83.8% (0%)**	99.6% (26%)
NN k4	89.6% (100%)	89.6% (100%)	91.9% (92%)
WC k5	87.9% (100%)	88.9% (100%)	96.6% (75%)
WC k91	94.9% (100%)	94.9% (100%)	100% (75%)
WC k114	89.8% (100%)	89.8% (100%)	93.4% (94%)
BFS k3	88.5% (100%)	88.5% (100%)	100% (100%)
BFS k9	**84.9% (0%)**	**84.9% (0%)**	86.1% (95%)
BFS k11	**82.1% (0%)**	**83.9% (0%)**	99.2% (90%)
KMN k1	**82.2% (0%)**	**82.6% (0%)**	100% (7%)
KMN k2	**81.0% (0%)**	85.4% (100%)	100% (7%)

* The resilience coverage requirement is set to be 85%.
* Kernels with no values in the fourth column only contain one fault distribution group, thus are not applicable for fine-grain analysis.

Coarse-grain Protection Overhead Analysis. We first show analysis at the kernel level (see the two columns under “Kernel-Level" in Table 8.4). We observe that resilience coverage increases as we start to relax the output quality requirement, which results in increasing overhead reduction (OR). For BlackScholes, for example, the resilience coverage is very low (31.8%) when users desire perfect output quality. With such low resilience coverage, it is necessary to protect the entire kernel (i.e., 0% overhead reduction). However, if users are able to accept some inaccuracy in output (i.e., accept the default output

quality), the resilience coverage increases to 89.1%. And if this is agreeable with the user, then the re-computation cost of the entire kernel can be avoided (i.e., 100% overhead reduction). In the remaining discussion, we assume a 85% resilience coverage requirement set by the user, as many kernels satisfy it at the default output quality threshold. For example, we find that for kernels such as RAY, Sort k8, and KMN k2, the resilience coverage requirement of 85% is met and hence its re-computation can be completely avoided leading to 100% reduction in protection overhead by accepting Default OQ instead the Perfect OQ. However, we also find some that other kernels (see cells in bold in Table 8.4) do not meet the 85% resilience coverage requirement even at the Default OQ. For such kernels, we have to resort to fine-grain analysis to seek opportunities of overhead reduction.

Fine-grain Protection Overhead Analysis. If the kernels do not meet the resilience coverage requirement, the protection overhead can still be reduced by exploiting the fact that some CTAs or warp groups are significantly more error-resilient than others (see Observations 8.6 and 8.7). We propose not to re-compute such groups and hence reduce the associated protection overhead. As CTAs are independent of each other, output of only those CTAs will be required to be compared that have lower resilience coverage. After applying our resilience characterization (Section 8.4), we find that the resilience coverage has increased significantly for most kernels (see “Default OQ (OR)" column under “CTA-/Warp-Level") and still with a significant overhead reduction. For example, for HotSpot, at the CTA level, users can obtain 99.6% resilience coverage while still reducing overhead by 26% (i.e., G1 in Figure 8.9(e) can be protection-free). In addition, for kernels with over 85% resilience coverage (i.e., NN k4, WC k5, WC k91, WC k114, and BFS k3), it is still possible to further improve their resilience coverage at a finer granularity (see fourth column). Although the above analysis is for the 85% resilience coverage requirement, similar analysis can be performed for any other threshold.

Effect of threshold values. So far, we have shown the trade-off between resilience coverage and overhead reduction using the default threshold values provided by the benchmarks. As different users have different resiliency requirements, it would be interesting to evaluate

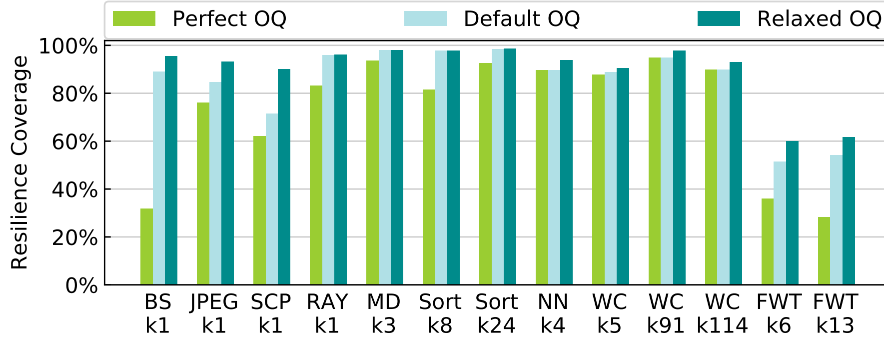


Figure 8.12: Resilience coverage (% of *Masked* + *SDC-accept* outputs) as a function of different output quality. Output quality changes with different SDC acceptability thresholds.

such trade-off under different threshold values. Here we further explore the trade-off under a relaxed output quality (i.e., evaluated with a relaxed threshold value as compare to the default one). Note that, determining the suitable relaxed threshold values are application-specific and beyond the scope of this preliminary study. Figure 8.12 shows the comparison in terms of the kernel-level resilience coverage.

We observe that, for some benchmark kernels, the resilience coverage increases as we start to relax the output quality requirement. For example, for BlackScholes K1, the resilience coverage increases from 89.1% under the Default OQ to 98.2% under the Relaxed OQ. In contrast, for other benchmarks, such as RAY K1 and WC K5, we observe little to no increase across different output qualities. This implies that different benchmark kernels show different sensitivity to the evaluating threshold values. Understanding the reason to this would help users determine a reasonable threshold value based on their resiliency requirements. Moreover, we can also explore the impact of different threshold values under finer granularities, such as CTAs and warps. We leave these as our future work.

Observation 8.8 *Hierarchical error-resilience analysis offers flexibility for resilience coverage and overhead reduction.*

8.6 Chapter Summary

In this chapter, we introduce the concept of *accuracy-aware resilience* for overhead reduction in the context of GPGPUs. We propose a hierarchical thread classification and selection approach to understand the application resilience coverage. Through a large number of fault injection runs ($330,000$ in total) on a variety of GPGPU applications, we obtain several interesting observations. First, the error resilience of GPGPU application kernels can significantly increase by embracing a reasonable loss in output quality. Second, the accuracy-aware error resilience of a kernel can be captured by analyzing threads of only a few thread-blocks. Third, the proposed hierarchical approach facilitates in reducing the overhead of protection/recovery mechanisms typically used by GPUs to ensure reliable output.

Chapter 9

Future Work

In this dissertation, we focus on studying the reliability of GPGPUs and conduct this analysis from two perspectives: system level and application level. At the system-level analysis, we study large-scale system logs on GPU errors in the field (see Chapters 3 and 4) and propose machine learning models to accurate predict error occurrences to enable low-overhead reliable computing at the system level (see Chapters 5 and 6). At the application-level, we build a progressive fault site pruning framework to systematically reduce the number of fault sites needed in a fault injection campaign (see Chapter 7). This framework is able to serve as an effective way to deepen the understanding on the error resilience characteristics for GPGPU applications. In Chapter 8, we hierarchically study the resilience features of GPGPU applications at various levels, including kernel, CTA, and warp levels. Here, we outline future work.

9.1 Fault Injection for Multiple Inputs

In Chapter 7, we propose an accurate and effective fault injection framework following GPGPU-specific features. This framework is application-dependent and input-dependent. The results shown in Chapter 7 are for one application input. We have to follow the four progressive pruning steps again for a different input. Even though the profiling cost is affordable (i.e., all information can be collected with one fault-free execution), it would

be interesting to study the trends and variations among the outcomes of fault injection experiments for different inputs. Li et al. [84] study the variation of silent data corruption (SDC) probabilities across different inputs of a program. By understanding the reasons of variations, they propose a model to bound the overall program SDC probabilities under multiple inputs of one application, with the result obtained by a single input. Similarly here, we would like to investigate the impact of multiple inputs for GPGPU applications. With current preliminary results, we observe that there are patterns across different inputs (i.e., different sizes and distributions) for benchmarks including HotSpot and PathFinder. Such observations would be helpful to infer the resiliency profile of one application under multiple inputs.

9.2 Low-Overhead Reliable GPU Computing

9.2.1 Thread-to-CTA Remapping

In Chapter 7, we observe that dynamic instruction count (short as *iCnt*) of a thread is an effective proxy for the error resilience profile of a thread. That is, threads with the same *iCnt* tend to share similar distribution of fault injection outcomes. The rationale is that threads with the same number of dynamic instruction count are likely to execute the same set of instructions and thus result in similar reactions to injected faults. Using this observation, it is natural to apply different levels of protection mechanisms to different threads. For example, using the commonly considered re-computation model [147, 39], we can duplicate the execution of threads that are sensitive to bit flips but execute only once the error-resilient threads, as distinguished by thread *iCnt*. Re-computational model requires comparing the output of two threads. If there is a mismatch, it could re-execute both threads to get the correct result [147]. Given that threads are grouped into CTAs for execution, it is more practical to design resilience solutions at the CTA level, instead of at the thread level.

Towards this perspective, the first step is to understand how threads with different

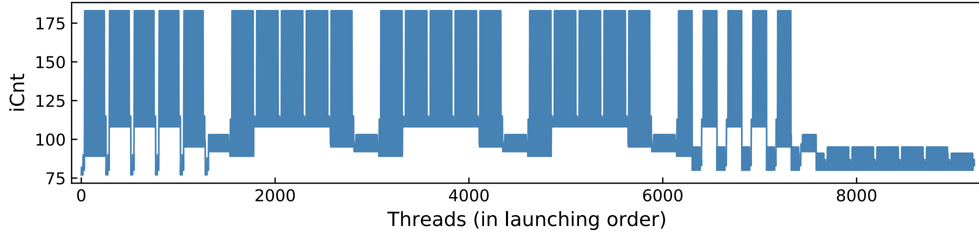


Figure 9.1: The dynamic instruction count of threads in their launching order for HotSpot..

iCnt are scheduled currently. Figure 9.1 shows the thread *iCnt* of each thread in their launching order for HotSpot. We observe that threads with different *iCnt* are executed together in consecutive time. Next, in Figure 9.2, we show the composition of threads with different *iCnt* inside each CTA for HotSpot. Each CTA in HotSpot contains 256 threads. However, none of the CTAs contains only one type of thread (i.e., threads with same *iCnt*). In fact, each CTA may contain 5 to 11 different types of threads, with one dominant thread type (marked with blue color in the figure). Figure 9.2 reveals that currently CTAs contain multiple thread types, so it is necessary to apply remapping. If for some other benchmarks, each CTA contains only one thread type, then there is no need for remapping. To summarize, for benchmarks like HotSpot, threads with the same *iCnt* are not scheduled together, making it hard to realize the aforementioned protection for different threads.

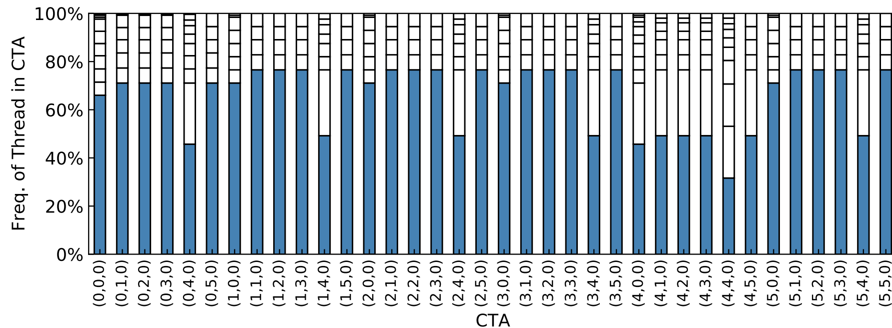


Figure 9.2: The composition of threads with different dynamic instruction count inside each CTA for HotSpot. Each CTA contains 256 threads. None of the CTAs contains only one thread type while the colored part represents the dominant thread type.

One solution is to break the current mapping of threads to CTAs and re-map threads

with same *iCnt* to the same CTA. That is, after re-mapping, each CTA only contains one type of thread. Consequently, we can apply the same protection (i.e., either duplicate the execution or relax the protection) at the CTA level. Clearly, this re-mapping can result in worse performance because it may break the data locality or other reasons. As long as the reduction in terms of error protection overhead is significant, it still makes such solution worthwhile and practical.

9.2.2 Resilience-Aware Scheduling

In the above section, we re-map the threads with same *iCnt* to the same CTA so that threads within the same CTA now share similar error resilience characteristics. This provides opportunities for developing resilience-aware scheduling algorithms for light-weight reliable GPU computing. In GPU systems, scheduling has been vastly studied at many levels (e.g., kernel, CTAs, and warps [67, 66, 71, 65, 73, 116, 72, 98]) in order to achieve better performance. For example, the two-level warp scheduling proposed by Narasiman et al. [98] increases core utilization by dividing the warps into smaller groups and scheduling them in a staggered manner. Rogers et al. [122] and Jog et al. [67] proposed warp schedulers to reduce contention in caches. Kayiran et al. [71] proposed to dynamically tune the thread-level parallelism. Lee et al. [81] proposed a criticality-aware warp scheduler that prefers critical warps over others for better latency tolerance. Adriaens et al. [20] and Pai et al. [114] proposed techniques to spatially partition the core resources across concurrent GPU kernels.

The next natural step is to design GPGPU resource scheduling algorithms for the purpose of low-overhead reliable computing. We will devise scheduling algorithms that consider *if the error protection of specific hardware components (spatial optimization) can be turned-off for a limited duration of the application execution (temporal optimization).* Recall that, a typical GPU consists of multiple simple cores, also called streaming-multiprocessors (SMs) in NVIDIA terminology [111]. For example, at an SM-level, we can develop kernel and CTA scheduling techniques such that error-resilient kernels and CTAs

are scheduled on SMs that are protection-free. We believe that even higher benefits can be achieved if scheduling techniques at all resource allocation levels (i.e., warps, CTAs, and kernels) are carefully designed.

We expect that side-effects of the resilience-aware scheduling could include performance- and load-imbalance across different compute resources (i.e., SMs). This is because of the fact that dynamic tuning of error protection would likely affect the runtime of certain warps, CTAs, and kernels differently. For example, if some SMs are fast (because they are now error protection free), we have to take care of possible indications of load imbalance. We will perform a detailed investigation to find such sources of performance degradation and develop techniques to mitigate them. Specifically, the SM-level criticality estimation proposed in [68] will help determine the amount of load imbalance in the system and design scheduling techniques at all levels of GPU hierarchy to alleviate the problem of load-imbalance.

Bibliography

[1] About CUDA. https://developer.nvidia.com/about-cuda.

[2] Blue Joule - BlueGene/Q, Power BQC 16C 1.60GHz, Custom. https://www.top500.org/system/177723.

[3] CUDA-GDB. http://docs.nvidia.com/cuda/cuda-gdb/#axzz4PHxjHEUB.

[4] GP100 Pascal Whitepaper. https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf.

[5] GPGPU-Sim Instruction Set Architecture.

[6] Gyoukou Supercomputer. https://www.top500.org/system/179102/.

[7] K20X By the Numbers. https://www.olcf.ornl.gov/for-users/system-user-guides/titan/nvidia-k20x-gpus/#k20x-by-the-numbers.

[8] NVIDIA Fermi Architecture Whitepaper. http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf.

[9] NVIDIA K20X GPUs. https://www.olcf.ornl.gov/for-users/system-user-guides/titan/nvidia-k20x-gpus/.

[10] NVIDIA Kepler GK110 Architecture Whitepaper. https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.

[11] Piz Daint Supercomputer. https://www.cscs.ch/computers/dismissed/piz-daint-piz-dora/.

[12] Sierra | Computation. https://computation.llnl.gov/computers/sierra.

[13] Summit – Oak Ridge Leadership Computing Facility. https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/.

[14] Tesla P100 Most Advanced Data Center Accelerator|NVIDIA. http://www.nvidia.com/object/tesla-p100.html.

[15] The Green500 List. https://www.top500.org/green500/lists/2017/11/.

[16] Tianhe-2 Supercomputer. http://en.nscc-gz.cn/Product/HighPerformanceComputingService/ServiceCharacteristics.html?

[17] Titan. https://www.olcf.ornl.gov/olcf-resources/compute-systems/titan/.

[18] Top500 List. https://www.top500.org/lists/2017/11/.

[19] Understanding XID Errors. http://docs.nvidia.com/deploy/xid-errors/index.html.

[20] Jacob Adriaens, Katherine Compton, Nam Sung Kim, and Michael J. Schulte. The case for GPGPU spatial multitasking. In *18th IEEE International Symposium on High Performance Computer Architecture, HPCA 2012, New Orleans, LA, USA, 25-29 February, 2012*, pages 79–90. IEEE Computer Society, 2012.

[21] Nesreen K Ahmed, Amir F Atiya, Neamat El Gayar, and Hisham El-Shishiny. An empirical comparison of machine learning models for time series forecasting. *Econometric Reviews*, 29(5-6):594–621, 2010.

[22] Artur Andrzejak and Luís Moura Silva. Using machine learning for non-intrusive modeling and prediction of software aging. In *IEEE/IFIP Network Operations and Management Symposium: Pervasive Management for Ubioquitous Networks and Services, NOMS 2008, 7-11 April 2008, Salvador, Bahia, Brazil*, Marcus

Brunner, Carlos Becker Westphall, and Lisandro Zambenedetti Granville, editors, pages 25–32. IEEE, 2008.

[23] Lakshmi Narayanan Bairavasundaram, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. *Characteristics, impact, and tolerance of partial disk failures.* PhD thesis, University of Wisconsin–Madison, 2008.

[24] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2009, April 26-28, 2009, Boston, Massachusetts, USA, Proceedings*, pages 163–174. IEEE Computer Society, 2009.

[25] Leonardo Arturo Bautista-Gomez, Franck Cappello, Luigi Carro, Nathan DeBardeleben, Bo Fang, Sudhanva Gurumurthi, Karthik Pattabiraman, Paolo Rech, and Matteo Sonza Reorda. Gpgpus: How to combine high computational power with high reliability. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2014, Dresden, Germany, March 24-28, 2014*, Gerhard P. Fettweis and Wolfgang Nebel, editors, pages 1–9. European Design and Automation Association, 2014.

[26] Robin M. Betz, Nathan DeBardeleben, and Ross C. Walker. An investigation of the effects of hard and soft errors on graphics processing unit-accelerated molecular dynamics simulations. *Concurrency and Computation: Practice and Experience*, 26(13):2134–2140, 2014.

[27] Mirela Madalina Botezatu, Ioana Giurgiu, Jasmina Bogojeska, and Dorothea Wiesmann. Predicting disk replacement towards reliable data centers. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, Balaji

Krishnapuram, Mohak Shah, Alexander J. Smola, Charu C. Aggarwal, Dou Shen, and Rajeev Rastogi, editors, pages 39–48. ACM, 2016.

[28] George EP Box, Gwilym M Jenkins, Gregory C Reinsel, and Greta M Ljung. *Time series analysis: forecasting and control*. John Wiley & Sons, 2015.

[29] Nitesh V Chawla. Data mining for imbalanced datasets: An overview. In *Data mining and knowledge discovery handbook*, pages 875–886. Springer, 2009.

[30] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. SMOTE: synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, 16:321–357, 2002.

[31] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization, IISWC 2009, October 4-6, 2009, Austin, TX, USA*, pages 44–54. IEEE Computer Society, 2009.

[32] Daniel Chen, Gabriela Jacques-Silva, Zbigniew Kalbarczyk, Ravishankar K Iyer, and Bruce Mealey. Error behavior comparison of multiple computing systems: A case study using linux on pentium, solaris on sparc, and aix on power. In *Dependable Computing, 2008. PRDC'08. 14th IEEE Pacific Rim International Symposium on*, pages 339–346. IEEE, 2008.

[33] Hyungmin Cho, Shahrzad Mirkhani, Chen-Yong Cher, Jacob A. Abraham, and Subhasish Mitra. Quantitative evaluation of soft error injection techniques for robust system design. In *The 50th Annual Design Automation Conference 2013, DAC '13, Austin, TX, USA, May 29 - June 07, 2013*, pages 101:1–101:10.

[34] PEZY Computing. Pezy-sc many core processor, 2014.

[35] Paulo Cortez, Miguel Rio, Miguel Rocha, and Pedro Sousa. Multi-scale internet traffic forecasting using neural networks and time series methods. *Expert Systems*, 29(2):143–155, 2012.

[36] Maria Couceiro, Paolo Romano, and Luís E. T. Rodrigues. A machine learning approach to performance prediction of total order broadcast protocols. In *Fourth IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2010, Budapest, Hungary, 27 September - 1 October 2010*, pages 184–193. IEEE Computer Society, 2010.

[37] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. The scalable heterogeneous computing (SHOC) benchmark suite. In *Proceedings of 3rd Workshop on General Purpose Processing on Graphics Processing Units, GPGPU 2010, Pittsburgh, Pennsylvania, USA, March 14, 2010*, David R. Kaeli and Miriam Leeser, editors, volume 425 of *ACM International Conference Proceeding Series*, pages 63–74. ACM, 2010.

[38] Sheng Di, Derrick Kondo, and Walfredo Cirne. Google hostload prediction based on bayesian model with optimized feature combination. *J. Parallel Distrib. Comput.*, 74(1):1820–1832, 2014.

[39] Martin Dimitrov, Mike Mantor, and Huiyang Zhou. Understanding software approaches for GPGPU reliability. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units, GPGPU 2009, Washington, DC, USA, March 8, 2009*, David R. Kaeli and Miriam Leeser, editors, volume 383 of *ACM International Conference Proceeding Series*, pages 94–104. ACM, 2009.

[40] Qia Ding. Long-term load forecast using decision tree method. *In Power Systems Conference and Exposition*, pages 1541–1543, 2006.

[41] Anders Eklund, Paul Dufort, Daniel Forsberg, and Stephen LaConte. Medical image processing on the GPU - past, present and future. *Medical Image Analysis*, 17(8):1073–1094, 2013.

[42] Nosayba El-Sayed and Bianca Schroeder. Reading between the lines of failure logs: Understanding how HPC systems fail. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Budapest, Hungary, June 24-27, 2013*, pages 1–12. IEEE Computer Society, 2013.

[43] Nosayba El-Sayed, Ioan A. Stefanovici, George Amvrosiadis, Andy A. Hwang, and Bianca Schroeder. Temperature management in data centers: why some (might) like it hot. In *ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '12, London, United Kingdom, June 11-15, 2012*, Peter G. Harrison, Martin F. Arlitt, and Giuliano Casale, editors, pages 163–174. ACM, 2012.

[44] Bo Fang, Karthik Pattabiraman, Matei Ripeanu, and Sudhanva Gurumurthi. Gpu-qin: A methodology for evaluating the error resilience of GPGPU applications. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2014, Monterey, CA, USA, March 23-25, 2014*, pages 221–230. IEEE Computer Society, 2014.

[45] N Farazmand, R Ubal, and D Kaeli. Statistical fault injection-based AVF analysis of a GPU architecture. *Proceedings of SELSE*, 2012.

[46] R Foster. How to harness big data for improving public health. *Government Health IT*, 2012.

[47] Vinicius Fratin, Daniel A. G. de Oliveira, Caio B. Lunardi, Fernando Santos, Gennaro Rodrigues, and Paolo Rech. Code-dependent and architecture-dependent reliability behaviors. In *48th Annual IEEE/IFIP Interna-*

tional Conference on Dependable Systems and Networks, DSN 2018, Luxembourg City, Luxembourg, June 25-28, 2018, pages 13–26.

[48] Haohuan Fu, Junfeng Liao, Jinzhe Yang, Lanning Wang, Zhenya Song, Xiaomeng Huang, Chao Yang, Wei Xue, Fangfang Liu, Fang-Li Qiao, Wei Zhao, Xunqiang Yin, Chaofeng Hou, Chenglong Zhang, Wei Ge, Jian Zhang, Yangang Wang, Chunbo Zhou, and Guangwen Yang. The Sunway TaihuLight supercomputer: system and applications. *SCIENCE CHINA Information Sciences*, 59(7):072001:1–072001:16, 2016.

[49] Song Fu and Cheng-Zhong Xu. Quantifying temporal and spatial correlation of failure events for proactive management. In *26th IEEE Symposium on Reliable Distributed Systems (SRDS 2007), Beijing, China, October 10-12, 2007*, pages 175–184. IEEE Computer Society, 2007.

[50] Ana Gainaru, Franck Cappello, Joshi Fullop, Stefan Trausan-Matu, and William Kramer. Adaptive event prediction strategy with dynamic time window for large-scale hpc systems. In *Managing Large-scale Systems via the Analysis of System Logs and the Application of Machine Learning Techniques*, page 4. ACM, 2011.

[51] Ana Gainaru, Franck Cappello, Marc Snir, and William Kramer. Fault prediction under the microscope: a closer look into HPC systems. In *SC Conference on High Performance Computing Networking, Storage and Analysis, SC '12, Salt Lake City, UT, USA - November 11 - 15, 2012*, Jeffrey K. Hollingsworth, editor, page 77. IEEE/ACM, 2012.

[52] Paul Goodwin et al. The holt-winters approach to exponential smoothing: 50 years old and going strong. *Foresight*, 19:30–33, 2010.

[53] Andreas W Gotz, Mark J Williamson, Dong Xu, Duncan Poole, Scott Le Grand, and Ross C Walker. Routine microsecond molecular dynamics sim-

ulations with amber on gpus. 1. generalized born. *Journal of chemical theory and computation*, 8(5):1542–1555, 2012.

[54] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. Auto-tuning a high-level language targeted to gpu codes. In *Innovative Parallel Computing (InPar), 2012*, pages 1–10. IEEE, 2012.

[55] Saurabh Gupta, Devesh Tiwari, Christopher Jantzi, James H. Rogers, and Don Maxwell. Understanding and exploiting spatial properties of system failures on extreme-scale HPC systems. In *45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2015, Rio de Janeiro, Brazil, June 22-25, 2015*, pages 37–44. IEEE Computer Society, 2015.

[56] Isabelle Guyon. A scaling law for the validation-set training-set size ratio. *AT&T Bell Laboratories*, 1997.

[57] Imran S. Haque and Vijay S. Pande. Hard data on soft errors: A large-scale assessment of real-world error rates in GPGPU. In *10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, CCGrid 2010, 17-20 May 2010, Melbourne, Victoria, Australia*, pages 691–696. IEEE Computer Society, 2010.

[58] Siva Kumar Sastry Hari, Sarita V. Adve, Helia Naeimi, and Pradeep Ramachandran. Relyzer: exploiting application-level fault equivalence to analyze application resiliency to transient faults. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3-7, 2012*, pages 123–134.

[59] Siva Kumar Sastry Hari, Timothy Tsai, Mark Stephenson, Stephen W. Keckler, and Joel S. Emer. SASSIFI: an architecture-level fault injection tool for GPU application resilience evaluation. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2017, Santa Rosa, CA, USA, April 24-25, 2017*, pages 249–258. IEEE, 2017.

[60] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. Mars: a mapreduce framework on graphics processors. In *17th International Conference on Parallel Architecture and Compilation Techniques, PACT 2008, Toronto, Ontario, Canada, October 25-29, 2008*, Andreas Moshovos, David Tarditi, and Kunle Olukotun, editors, pages 260–269. ACM, 2008.

[61] Liang Hu, Xilong Che, and Si-Qing Zheng. Online system for grid resource monitoring and machine learning-based prediction. *IEEE Trans. Parallel Distrib. Syst.*, 23(1):134–145, 2012.

[62] Andy A. Hwang, Ioan A. Stefanovici, and Bianca Schroeder. Cosmic rays don't strike twice: understanding the nature of DRAM errors and the implications for system design. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3-7, 2012*, Tim Harris and Michael L. Scott, editors, pages 111–122. ACM, 2012.

[63] Rabindra Ku Jena, Musbah M Aqel, Pankaj Srivastava, and Prabhat K Mahanti. Soft computing methodologies in bioinformatics. *European Journal of Scientific Research*, 2009.

[64] Hyeran Jeon, Mark Wilkening, Vilas Sridharan, Sudhanva Gurumurthi, and G Loh. Architectural vulnerability modeling and analysis of integrated graphics processors. In *Workshop on Silicon Errors in Logic-System Effects (SELSE), Stanford, CA*, 2012.

[65] Adwait Jog, Evgeny Bolotin, Zvika Guz, Mike Parker, Stephen W. Keckler, Mahmut T. Kandemir, and Chita R. Das. Application-aware memory system for fair and efficient execution of concurrent GPGPU applications. In *Seventh Workshop on General Purpose Processing Using GPUs, GPGPU-7, Salt Lake*

City, UT, USA, March 1, 2014, John Cavazos, Xiang Gong, and David R. Kaeli, editors, page 1. ACM, 2014.

[66] Adwait Jog, Onur Kayiran, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das. Orchestrated scheduling and prefetching for gpgpus. In *The 40th Annual International Symposium on Computer Architecture, ISCA'13, Tel-Aviv, Israel, June 23-27, 2013*, Avi Mendelson, editor, pages 332–343. ACM, 2013.

[67] Adwait Jog, Onur Kayiran, Nachiappan Chidambaram Nachiappan, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das. OWL: cooperative thread array aware scheduling techniques for improving GPGPU performance. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*, Vivek Sarkar and Rastislav Bodík, editors, pages 395–406. ACM, 2013.

[68] Adwait Jog, Onur Kayiran, Ashutosh Pattnaik, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das. Exploiting core criticality for enhanced GPU performance. In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science, Antibes Juan-Les-Pins, France, June 14-18, 2016*, Sara Alouf, Alain Jean-Marie, Nidhi Hegde, and Alexandre Proutière, editors, pages 351–363. ACM, 2016.

[69] Wayne Joubert, Douglas Kothe, et al. Preparing for exascale: ORNL leadership computing facility application requirements and strategy. *ORNL Technical Report*, 2009.

[70] Manolis Kaliorakis, Dimitris Gizopoulos, Ramon Canal, and Antonio González. Merlin: Exploiting dynamic instruction behavior for fast and accurate microarchitecture level reliability assessment. In *Proceedings of the 44th An-*

nual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017, pages 241–254. ACM, 2017.

[71] Onur Kayiran, Adwait Jog, Mahmut T. Kandemir, and Chita R. Das. Neither more nor less: Optimizing thread-level parallelism for gpgpus. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques, Edinburgh, United Kingdom, September 7-11, 2013*, Christian Fensch, Michael F. P. O'Boyle, André Seznec, and François Bodin, editors, pages 157–166. IEEE Computer Society, 2013.

[72] Onur Kayiran, Adwait Jog, Ashutosh Pattnaik, Rachata Ausavarungnirun, Xulong Tang, Mahmut T. Kandemir, Gabriel H. Loh, Onur Mutlu, and Chita R. Das. μc-states: Fine-grained GPU datapath power management. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation, PACT 2016, Haifa, Israel, September 11-15, 2016*, Ayal Zaks, Bilha Mendelson, Lawrence Rauchwerger, and Wen-mei W. Hwu, editors, pages 17–30. ACM, 2016.

[73] Onur Kayiran, Nachiappan Chidambaram Nachiappan, Adwait Jog, Rachata Ausavarungnirun, Mahmut T. Kandemir, Gabriel H. Loh, Onur Mutlu, and Chita R. Das. Managing GPU concurrency in heterogeneous architectures. In *47th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2014, Cambridge, United Kingdom, December 13-17, 2014*, pages 114–126. IEEE Computer Society, 2014.

[74] Dong-Wan Kim and Mattan Erez. Balancing reliability, cost, and performance tradeoffs with freefault. In *21st IEEE International Symposium on High Performance Computer Architecture, HPCA 2015, Burlingame, CA, USA, February 7-11, 2015*, pages 439–450. IEEE Computer Society, 2015.

[75] Jungrae Kim, Michael Sullivan, and Mattan Erez. Bamboo ECC: strong, safe, and flexible codes for reliable computer memory. In *21st IEEE International Symposium on High Performance Computer Architecture, HPCA 2015, Burlingame, CA, USA, February 7-11, 2015*, pages 101–112. IEEE Computer Society, 2015.

[76] Jungrae Kim, Michael Sullivan, Seong-Lyong Gong, and Mattan Erez. Frugal ECC: efficient and versatile memory error protection through fine-grained compression. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, Austin, TX, USA, November 15-20, 2015*, Jackie Kern and Jeffrey S. Vetter, editors, pages 12:1–12:12. ACM, 2015.

[77] David B Kirk and W Hwu Wen-Mei. *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann, 2016.

[78] Richard Koo and Sam Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Trans. Software Eng.*, 13(1):23–31, 1987.

[79] Douglas Kothe and Ricky Kendall. Computational science requirements for leadership computing. *ORNL Technical Report*, 2007.

[80] Supada Laosooksathit, Nichamon Naksinehaboon, Chokchai Leangsuksan, Apurba Dhungana, Clayton Chandler, Kasidit Chanchio, and Amir Farbin. Lightweight checkpoint mechanism and modeling in GPGPU environment. *Computing (HPC Syst)*, 12:13–20, 2010.

[81] Shin-Ying Lee and Carole-Jean Wu. CAWS: criticality-aware warp scheduling for GPGPU workloads. In *International Conference on Parallel Architectures and Compilation, PACT '14, Edmonton, AB, Canada, August 24-27, 2014*, José Nelson Amaral and Josep Torrellas, editors, pages 175–186. ACM, 2014.

[82] Lawrence M Leemis and Stephen Keith Park. *Discrete-event simulation: A first course*. Pearson Prentice Hall Upper Saddle River, NJ, 2006.

[83] Régis Leveugle, A. Calvez, Paolo Maistri, and Pierre Vanhauwaert. Statistical fault injection: Quantified error and confidence. In *Design, Automation and Test in Europe, DATE 2009, Nice, France, April 20-24, 2009*, Luca Benini, Giovanni De Micheli, Bashir M. Al-Hashimi, and Wolfgang Müller, editors. IEEE, 2009.

[84] Guanpeng Li and Karthik Pattabiraman. Modeling input-dependent error propagation in programs. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 279–290. IEEE, 2018.

[85] Guanpeng Li, Karthik Pattabiraman, Chen-Yong Cher, and Pradip Bose. Understanding error propagation in GPGPU applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Salt Lake City, UT, USA, November 13-18, 2016*, John West and Cherri M. Pancake, editors, pages 240–251. IEEE Computer Society, 2016.

[86] Jia Li and Andrew W Moore. Forecasting web page views: methods and observations. *Journal of Machine Learning Research*, 9(Oct):2217–2250, 2008.

[87] Jing Li, Xinpu Ji, Yuhan Jia, Bingpeng Zhu, Gang Wang, Zhongwei Li, and Xiaoguang Liu. Hard drive failure prediction using classification and regression trees. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*, pages 383–394. IEEE Computer Society, 2014.

[88] Yinglung Liang, Yanyong Zhang, Anand Sivasubramaniam, Morris Jette, and Ramendra K. Sahoo. Bluegene/l failure analysis and prediction models. In *2006 International Conference on Dependable Systems and Networks*

(DSN 2006), 25-28 June 2006, Philadelphia, Pennsylvania, USA, Proceedings, pages 425–434. IEEE Computer Society, 2006.

[89] Yinglung Liang, Yanyong Zhang, Anand Sivasubramaniam, Ramendra K. Sahoo, José E. Moreira, and Manish Gupta. Filtering failure logs for a BlueGene/L prototype. In *2005 International Conference on Dependable Systems and Networks (DSN 2005), 28 June - 1 July 2005, Yokohama, Japan, Proceedings*, pages 476–485. IEEE Computer Society, 2005.

[90] Robert Lucas. Top ten exascale research challenges. In *DOE ASCAC Subcommittee Report*, 2014.

[91] Farzaneh Mahdisoltani, Ioan A. Stefanovici, and Bianca Schroeder. Proactive error prediction to improve storage system reliability. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017.*, pages 391–402. USENIX Association, 2017.

[92] Catello Di Martino, Zbigniew T. Kalbarczyk, Ravishankar K. Iyer, Fabio Baccanico, Joseph Fullop, and William Kramer. Lessons learned from the analysis of system failures at petascale: The case of blue waters. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*, pages 610–621. IEEE Computer Society, 2014.

[93] Celso L. Mendes, Brett Bode, Gregory H. Bauer, Jeremy Enos, Cristina Beldica, and William T. Kramer. Deploying a large petascale system: The blue waters experience. In *Proceedings of the International Conference on Computational Science, ICCS 2014, Cairns, Queensland, Australia, 10-12 June, 2014*, David Abramson, Michael Lees, Valeria V. Krzhizhanovskaya, Jack J. Dongarra, and Peter M. A. Sloot, editors, volume 29 of *Procedia Computer Science*, pages 198–209. Elsevier, 2014.

[94] Jiayuan Meng, Srimat T. Chakradhar, and Anand Raghunathan. Best-effort parallel execution framework for recognition and mining applications. In *23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, May 23-29, 2009*, pages 1–12. IEEE, 2009.

[95] Justin Meza et al. A large-scale study of flash memory errors in the field. *ACM SIGMETRICS Performance Evaluation Review*, 2015.

[96] Sushmita Mitra and Yoichi Hayashi. Bioinformatics with soft computing. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 36(5):616–635, 2006.

[97] Sushmita Mitra, Sankar K. Pal, and Pabitra Mitra. Data mining in soft computing framework: a survey. *IEEE Trans. Neural Networks*, 13(1):3–14, 2002.

[98] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N. Patt. Improving GPU performance via large warps and two-level warp scheduling. In *44rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2011, Porto Alegre, Brazil, December 3-7, 2011*, Carlo Galuzzi, Luigi Carro, Andreas Moshovos, and Milos Prvulovic, editors, pages 308–317. ACM, 2011.

[99] Iyswarya Narayanan, Di Wang, Myeongjae Jeon, Bikash Sharma, Laura Caulfield, Anand Sivasubramaniam, Ben Cutler, Jie Liu, Badriddine M. Khessib, and Kushagra Vaid. SSD failures in datacenters: What, when and why? In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science, Antibes Juan-Les-Pins, France, June 14-18, 2016*, Sara Alouf, Alain Jean-Marie, Nidhi Hegde, and Alexandre Proutière, editors, pages 407–408. ACM, 2016.

[100] Bin Nie, Adwait Jog, and Evgenia Smirni. A Hierarchical Approach to Enabling Low-Overhead Reliable GPU Computing. Undersubmission.

[101] Bin Nie, Devesh Tiwari, Saurabh Gupta, Evgenia Smirni, and James H. Rogers. A large-scale study of soft-errors on gpus in the field. In *2016 IEEE International Symposium on High Performance Computer Architecture, HPCA 2016, Barcelona, Spain, March 12-16, 2016*, pages 519–530. IEEE Computer Society, 2016.

[102] Bin Nie, Ji Xue, Saurabh Gupta, Christian Engelmann, Evgenia Smirni, and Devesh Tiwari. Characterizing temperature, power, and soft-error behaviors in data center systems: Insights, challenges, and opportunities. In *25th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, MASCOTS 2017, Banff, AB, Canada, September 20-22, 2017*, pages 22–31. IEEE Computer Society, 2017.

[103] Bin Nie, Ji Xue, Saurabh Gupta, Tirthak Patel, Christian Engelmann, Evgenia Smirni, and Devesh Tiwari. Machine learning models for GPU error prediction in a large scale HPC systems. In *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2018, Luxemburg City, Luxemburg, June 25-28, 2018*. IEEE Computer Society, 2018.

[104] Bin Nie, Lishan Yang, Adwait Jog, and Evgenia Smirni. Fault Site Pruning for Practical Reliability Analysis of GPGPU Applications. In *MICRO*, 2018.

[105] Ali Yadavar Nikravesh, Samuel A. Ajila, and Chung-Horng Lung. Towards an autonomic auto-scaling prediction system for cloud resource provisioning. In *10th IEEE/ACM International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2015, Florence, Italy, May 18-19, 2015*, Paola Inverardi and Bradley R. Schmerl, editors, pages 35–45. IEEE Computer Society, 2015.

[106] NVIDIA. Computational Finance. http://www.nvidia.com/object/computational_finance.html.

[107] NVIDIA. CUDA C programming guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide.

[108] NVIDIA. Dynamic page retirement. http://docs.nvidia.com/deploy/dynamic-page-retirement/index.html.

[109] NVIDIA. Researchers Deploy GPUs to Build World's Largest Artificial Neural Network. https://nvidianews.nvidia.com/news/researchers-deploy-gpus-to-build-world-s-largest-artificial-neural-network.

[110] NVIDIA. CUDA C/C++ SDK Code Samples. http://developer.nvidia.com/cuda-cc-sdk-code-samples, 2011.

[111] NVIDIA. Fermi: NVIDIA's Next Generation CUDA Compute Architecture, 2011.

[112] NVIDIA. JP Morgan speeds risk calculations with NVIDIA GPUs. 2011.

[113] Adam J. Oliner and Jon Stearley. What supercomputers say: A study of five system logs. In *The 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2007, 25-28 June 2007, Edinburgh, UK, Proceedings*, pages 575–584. IEEE Computer Society, 2007.

[114] Sreepathi Pai, Matthew J. Thazhuthaveetil, and R. Govindarajan. Improving GPGPU concurrency with elastic kernels. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*, Vivek Sarkar and Rastislav Bodík, editors, pages 407–418. ACM, 2013.

[115] Michael K Patterson. The effect of data center temperature on energy efficiency. In *Thermal and Thermomechanical Phenomena in Electronic Systems, 2008. ITHERM 2008. 11th Intersociety Conference on*, pages 1167–1174. IEEE, 2008.

[116] Ashutosh Pattnaik, Xulong Tang, Adwait Jog, Onur Kayiran, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, and Chita R. Das. Scheduling

techniques for GPU architectures with processing-in-memory capabilities. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation, PACT 2016, Haifa, Israel, September 11-15, 2016*, Ayal Zaks, Bilha Mendelson, Lawrence Rauchwerger, and Wen-mei W. Hwu, editors, pages 31–44. ACM, 2016.

[117] Antonio Pecchia, Domenico Cotroneo, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Improving log-based field failure data analysis of multi-node computing systems. In *Proceedings of the 2011 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2011, Hong Kong, China, June 27-30 2011*, pages 97–108. IEEE Compute Society, 2011.

[118] David Martin Powers. Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation. 2011.

[119] Guillem Pratx and Lei Xing. GPU computing in medical physics: A review. *Medical physics*, 38(5):2685–2697, 2011.

[120] Foster Provost. Machine learning from imbalanced data sets 101. In *Proceedings of the AAAI'2000 workshop on imbalanced data sets*, pages 1–3, 2000.

[121] Gonzalo Pedro Rodrigo Álvarez, Per-Olov Östberg, Erik Elmroth, Katie Antypas, Richard Gerber, and Lavanya Ramakrishnan. Hpc system lifetime story: Workload characterization and evolutionary analyses on nersc systems. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pages 57–60. ACM, 2015.

[122] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. Cache-conscious wavefront scheduling. In *45th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2012, Vancouver, BC, Canada, December 1-5, 2012*, pages 72–83. IEEE Computer Society, 2012.

[123] Ramendra K. Sahoo, Anand Sivasubramaniam, Mark S. Squillante, and Yanyong Zhang. Failure data analysis of a large-scale heterogeneous server environment. In *2004 International Conference on Dependable Systems and Networks (DSN 2004), 28 June - 1 July 2004, Florence, Italy, Proceedings*, page 772. IEEE Computer Society, 2004.

[124] Behrooz Sangchoolie, Karthik Pattabiraman, and Johan Karlsson. One bit is (not) enough: An empirical study of the impact of single and multiple bit-flip errors. In *47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2017, Denver, CO, USA, June 26-29, 2017*, pages 97–108. IEEE Computer Society, 2017.

[125] I Schmerken. Wall street accelerates options analysis with GPU technology. *Wall Street Technology*, 11, 2009.

[126] Bianca Schroeder and Garth A. Gibson. Disk failures in the real world: What does an MTTF of 1, 000, 000 hours mean to you? In *5th USENIX Conference on File and Storage Technologies, FAST 2007, February 13-16, 2007, San Jose, CA, USA*, Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau, editors, pages 1–16. USENIX, 2007.

[127] Bianca Schroeder and Garth A Gibson. Understanding failures in petascale computers. In *Journal of Physics: Conference Series*, volume 78, page 012022. IOP Publishing, 2007.

[128] Bianca Schroeder and Garth A. Gibson. A large-scale study of failures in high-performance computing systems. *IEEE Trans. Dependable Sec. Comput.*, 7(4):337–351, 2010.

[129] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM errors in the wild: a large-scale field study. In *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems,*

SIGMETRICS/Performance 2009, Seattle, WA, USA, June 15-19, 2009, John R. Douceur, Albert G. Greenberg, Thomas Bonald, and Jason Nieh, editors, pages 193–204. ACM, 2009.

[130] Milad Shokouhi. Detecting seasonal queries by time-series analysis. In *Proceeding of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 2011, Beijing, China, July 25-29, 2011*, Wei-Ying Ma, Jian-Yun Nie, Ricardo A. Baeza-Yates, Tat-Seng Chua, and W. Bruce Croft, editors, pages 1171–1172. ACM, 2011.

[131] Taniya Siddiqua, Athanasios E Papathanasiou, Arijit Biswas, and Sudhanva Gurumurthi. Analysis and modeling of memory errors from large-scale field data collection. In *SELSE*, 2013.

[132] Ruben Sipos, Dmitriy Fradkin, Fabian Mörchen, and Zhuang Wang. Log-based predictive maintenance. In *The 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '14, New York, NY, USA - August 24 - 27, 2014*, Sofus A. Macskassy, Claudia Perlich, Jure Leskovec, Wei Wang, and Rayid Ghani, editors, pages 1867–1876. ACM, 2014.

[133] Marc Snir, Robert W. Wisniewski, Jacob A. Abraham, Sarita V. Adve, Saurabh Bagchi, Pavan Balaji, Jim Belak, Pradip Bose, Franck Cappello, Bill Carlson, Andrew A. Chien, Paul Coteus, Nathan DeBardeleben, Pedro C. Diniz, Christian Engelmann, Mattan Erez, Saverio Fazzari, Al Geist, Rinku Gupta, Fred Johnson, Sriram Krishnamoorthy, Sven Leyffer, Dean Liberty, Subhasish Mitra, Todd Munson, Rob Schreiber, Jon Stearley, and Eric Van Hensbergen. Addressing failures in exascale computing. *IJHPCA*, 28(2):129–173, 2014.

[134] Vilas Sridharan and Dean Liberty. A study of DRAM failures in the field. In *SC Conference on High Performance Computing Networking, Storage and Analysis,*

SC '12, Salt Lake City, UT, USA - November 11 - 15, 2012, Jeffrey K. Hollingsworth, editor, page 76. IEEE/ACM, 2012.

[135] Vilas Sridharan, Jon Stearley, Nathan DeBardeleben, Sean Blanchard, and Sudhanva Gurumurthi. Feng shui of supercomputer memory: positional effects in DRAM and SRAM faults. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC'13, Denver, CO, USA - November 17 - 21, 2013*, William Gropp and Satoshi Matsuoka, editors, pages 22:1–22:11. ACM, 2013.

[136] Sam S. Stone, Justin P. Haldar, Stephanie C. Tsao, Wen-mei W. Hwu, Bradley P. Sutton, and Zhi-Pei Liang. Accelerating advanced MRI reconstructions on gpus. *J. Parallel Distrib. Comput.*, 68(10):1307–1318, 2008.

[137] Hiroyuki Takizawa, Katsuto Sato, Kazuhiko Komatsu, and Hiroaki Kobayashi. CheCUDA: A checkpoint/restart tool for CUDA applications. In *2009 International Conference on Parallel and Distributed Computing, Applications and Technologies, PDCAT 2009, Higashi Hiroshima, Japan, 8-11 December 2009*, pages 408–413.

[138] Jingweijia Tan, Nilanjan Goswami, Tao Li, and Xin Fu. Analyzing soft-error vulnerability on GPGPU microarchitecture. In *Proceedings of the 2011 IEEE International Symposium on Workload Characterization, IISWC 2011, Austin, TX, USA, November 6-8, 2011*, pages 226–235. IEEE Computer Society, 2011.

[139] Devesh Tiwari, Saurabh Gupta, George Gallarno, Jim Rogers, and Don Maxwell. Reliability lessons learned from GPU experience with the titan supercomputer at oak ridge leadership computing facility. pages 38:1–38:12, 2015.

[140] Devesh Tiwari, Saurabh Gupta, James H. Rogers, Don Maxwell, Paolo Rech, Sudharshan S. Vazhkudai, Daniel A. G. de Oliveira, Dave

Londo, Nathan DeBardeleben, Philippe Olivier Alexandre Navaux, Luigi Carro, and Arthur S. Bland. Understanding GPU errors on large-scale HPC systems and the implications for system design and operation. In *21st IEEE International Symposium on High Performance Computer Architecture, HPCA 2015, Burlingame, CA, USA, February 7-11, 2015*, pages 331–342. IEEE Computer Society, 2015.

[141] Devesh Tiwari, Saurabh Gupta, and Sudharshan S. Vazhkudai. Lazy checkpointing: Exploiting temporal locality in failures to mitigate checkpointing overheads on extreme-scale systems. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*, pages 25–36. IEEE Computer Society, 2014.

[142] Nancy Tran and Daniel A. Reed. Automatic ARIMA time series modeling for adaptive I/O prefetching. *IEEE Trans. Parallel Distrib. Syst.*, 15(4):362–377, 2004.

[143] Hong Linh Truong and Thomas Fahringer. Soft computing approach to performance analysis of parallel and distributed programs. 3648:50–60, 2005.

[144] Sotiris Tselonis and Dimitris Gizopoulos. GUFI: A framework for gpus reliability assessment. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2016, Uppsala, Sweden, April 17-19, 2016*, pages 90–100. IEEE Computer Society, 2016.

[145] Radha Venkatagiri, Abdulrahman Mahmoud, Siva Kumar Sastry Hari, and Sarita V. Adve. Approxilyzer: Towards a systematic framework for instruction-level approximate computing and its application to hardware resiliency. In *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*, pages 42:1–42:14. IEEE Computer Society, 2016.

[146] Jeffrey S. Vetter, Richard Glassbrook, Jack J. Dongarra, Karsten Schwan, Bruce Loftis, Stephen McNally, Jeremy S. Meredith, James H. Rogers, Philip C. Roth, Kyle Spafford, and Sudhakar Yalamanchili. Keeneland: Bringing heterogeneous GPU computing to the computational science community. *Computing in Science and Engineering*, 13(5):90–95, 2011.

[147] Jack Wadden, Alexander Lyashevsky, Sudhanva Gurumurthi, Vilas Sridharan, and Kevin Skadron. Real-world design and evaluation of compiler-managed GPU redundant multithreading. In *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*, pages 73–84. IEEE Computer Society, 2014.

[148] William Webber, Alistair Moffat, and Justin Zobel. A similarity measure for indefinite rankings. *ACM Trans. Inf. Syst.*, 28(4):20:1–20:38, 2010.

[149] Ji Xue, Robert Birke, Lydia Y. Chen, and Evgenia Smirni. Spatial-temporal prediction models for active ticket managing in data centers. *IEEE Trans. Network and Service Management*, 15(1):39–52, 2018.

[150] Ji Xue, Bin Nie, and Evgenia Smirni. Fill-in the gaps: Spatial-temporal models for missing data. In *13th International Conference on Network and Service Management, CNSM 2017, Tokyo, Japan, November 26-30, 2017*, pages 1–9. IEEE Computer Society, 2017.

[151] Ji Xue, Feng Yan, Robert Birke, Lydia Y. Chen, Thomas Scherer, and Evgenia Smirni. PRACTISE: robust prediction of data center time series. In *11th International Conference on Network and Service Management, CNSM 2015, Barcelona, Spain, November 9-13, 2015*, Mauro Tortonesi, Jürgen Schönwälder, Edmundo Roberto Mauro Madeira, Corinna Schmitt, and Joan Serrat, editors, pages 126–134. IEEE Computer Society, 2015.

[152] Ji Xue, Feng Yan, Alma Riska, and Evgenia Smirni. Proactive management of systems via hybrid analytic techniques. In *2015 International Conference on Cloud and Autonomic Computing, Boston, MA, USA, September 21-25, 2015*, pages 137–148. IEEE Computer Society, 2015.

[153] Keun Soo Yim, Zbigniew Kalbarczyk, and Ravishankar K Iyer. Measurement-based analysis of fault and error sensitivities of dynamic memory. In *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*, pages 431–436. IEEE, 2010.

[154] Keun Soo Yim, Cuong Manh Pham, Mushfiq Saleheen, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Hauberk: Lightweight silent data corruption error detector for GPGPU. In *25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011, Anchorage, Alaska, USA, 16-20 May, 2011 - Conference Proceedings*, pages 287–300. IEEE, 2011.

[155] Doe Hyun Yoon and Mattan Erez. Memory mapped ECC: low-cost error protection for last level caches. In *36th International Symposium on Computer Architecture (ISCA 2009), June 20-24, 2009, Austin, TX, USA*, Stephen W. Keckler and Luiz André Barroso, editors, pages 116–127. ACM, 2009.

[156] Doe Hyun Yoon and Mattan Erez. Virtualized and flexible ECC for main memory. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2010, Pittsburgh, Pennsylvania, USA, March 13-17, 2010*, James C. Hoe and Vikram S. Adve, editors, pages 397–408. ACM, 2010.

[157] Shenglin Zhang, Ying Liu, Weibin Meng, Zhiling Luo, Jiahao Bu, Sen Yang, Peixian Liang, Dan Pei, Jun Xu, Yuzhi Zhang, Yu Chen, Hui Dong, Xianping Qu, and Lei Song. Prefix: Switch failure prediction in datacenter networks. *POMACS*, 2(1):2:1–2:29, 2018.

[158] Zhenyun Zhuang, Haricharan Ramachandra, Cuong Tran, Subbu Subramaniam, Chavdar Botev, Chaoyue Xiong, and Badri Sridharan. Capacity planning and headroom analysis for taming database replication latency: Experiences with linkedin internet traffic. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering, Austin, TX, USA, January 31 - February 4, 2015*, Lizy K. John, Connie U. Smith, Kai Sachs, and Catalina M. Lladó, editors, pages 39–50. ACM, 2015.