THE UNIVERSITY
of EDINBURGH

This thesis has been submitted in fulfilment of the requirements for a postgraduate degree (e.g. PhD, MPhil, DClinPsychol) at the University of Edinburgh. Please note the following terms and conditions of use:

- This work is protected by copyright and other intellectual property rights, which are retained by the thesis author, unless otherwise stated.
- A copy can be downloaded for personal non-commercial research or study, without prior permission or charge.
- This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the author.
- The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the author.
- When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given.

Characterizing and Exploiting Application Behavior Under Data Corruption

Georgios Stefanakis

Master of Philosophy

Institute of Computing Systems Architecture
School of Informatics
University of Edinburgh

2015

Abstract

Shrinking semiconductor technologies come at the cost of higher susceptibility to hardware faults that render the systems unreliable. Traditionally, reliability solutions are aimed to protect equally and exhaustively all hardware parts of a system. This is in order to maintain the illusion of a correctly operating hardware. Due to the increasing error rates that induce higher reliability costs, this approach can no longer be sustainable.

It is a fact that hardware faults can be masked by various levels of fault-masking effects. Therefore, not all hardware faults manifest as the same outcome on an application's execution. Motivated by this fact, we propose a shift to vulnerability-driven unequal protection of a given structure (or same-level structures), where the less-vulnerable parts of a structure are protected less than their more-vulnerable counterparts.

For that purpose, in this thesis, we quantitatively investigate how the effect of hardware-induced data corruptions on application behavior varies. We develop a portable software-implemented fault-injection (SWIFI) tool. On top of performing single-bit fault injections to capture their effects on application behavior, our tool is also data-level aware and tracks the corrupted data to obtain more of their characteristics. This enables to analyze the effects of single-bit data corruptions in relation to the corrupted data characteristics and the executing workload. After a set of extensive fault-injection experiments on programs from the NAS Parallel Benchmarks suite, we obtain detailed insight on how the vulnerability varies; among others, for different application data types and for different bit locations within the data.

The results show that we can characterize the vulnerability of data based on their high-level characteristics (e.g. usage type, size, user and memory space location). Moreover, we conclude that application data are vulnerable in parts. All these show that there is potential in exploiting the application behavior under data corruption. The exhaustive equal protection can be avoided by safely shifting to vulnerability-driven unequal protection within given structures. This can reduce the reliability overheads, without a significant impact on the fault coverage. For that purpose, we demonstrate the potential benefits of exploiting the varying vulnerability characteristics of application data in the case of a data cache.

Lay Summary

Computer applications are executing on computer hardware and manipulate data stored in the computer hardware to perform computations and decisions according to their intended functionality. Unfortunately the underlying computer hardware is becoming increasingly unreliable due to a variety of physical effects. Such occurrences of hardware faults may unpredictably change the values of the application data causing the computer applications to potentially have unexpected behavior.

Interestingly though, the hardware faults do not affect the executing application always in the same manner; there are many different effects that a hardware fault may have on the behavior of an application. E.g., the application may stop operations, it may complete but compute a wrong output, it may complete successfully, etc.

One of the goals of this thesis is to investigate *how an application behaves under the presence of hardware faults that corrupt the application data*. For that purpose we develop a framework that emulates hardware faults in application data during the execution of an application and captures the resulting behavior of the application. After performing extensive tests on a set of applications, among other things, we can observe which application data are less likely than others to affect the application behavior if corrupted.

Based on this vulnerability insight, we propose that we can *exploit the application data vulnerabilities* to our benefit. As it is imperative that computing systems behave correctly, it has been common to protect fully the computing systems against as many hardware faults as possible. Such exhaustive protection though comes at the cost of increased operating costs and decreased performance in the effort to maximize the system's reliability.

As we find out that not all application data are equally vulnerable, we propose that it is not necessary to fully protect a computing system. Instead we can unequally protect a system by decreasing the protection of the areas that are less vulnerable if a hardware fault occurs there. That way we can reduce the protection overheads with a minimum effect on the system's reliability. As a mean to motivate such a design, we demonstrate the magnitude of potential benefits of unequal protection in the case of one hardware structure in particular, a data cache.

Acknowledgements

First, I would like to acknowledge the contribution of my supervisors, Marcelo Cintra and Vijay Nagarajan, to this work and to my personal development. Without their effort and guidance, this work wouldn't have been possible and, without their presence, I wouldn't have got where I am today. It is because of them that I got the opportunity to gain valuable research skills and life experience on the inner workings of the academic community.

I would also like to thank the examiners, Boris Grot and Yannis Andreopoulos, for their valuable comments and constructive suggestions on improving the quality of this thesis. Equally important were the rest members of the institute. In particular, I would like to thank the ICSA Directors, Murray Cole and Mike O'Boyle, for all their help and support. Also, my kindest regards go to all the fellow students of the group and to the fellow New Texas residents over the years. Many thanks to all of you Andrew, Fabricio, Karthik, Luna, Murali and Vassilis, who slowly became Kiran, Praveen, Rui, Stan and Ursula.

An extra special thank you goes to all of those that our time overlapped in Edinburgh and got close enough to consider them my friends, made my time in Edinburgh more fun, were kind enough to tolerate me and kept me sane too. Here's a big one for Andrew, George, Konstantina, Lito, Natasa, Nikolas, Salvatore and Vassilis. The same goes for all of those that have kept in touch with me over the years. I may not mention you by name but I am nevertheless very grateful for being there for me and remaining friends throughout the years.

This would not be a proper acknowledgement section without a hello to Jason Isaacs and a shout-out to the EPT/PDT/ETT members. Alexi, Anastasia, Evripidi and Stavre, there is a state-sized thank you coming your way. I cannot picture life without your wisdom, wittiness, bickering, wittering, reposting and five nines availability.

Before concluding this list, I want to state my gratitude to my parents for their support. Last but not least, I would like to thank my dear brother, Filippos, for coping, for being understanding and for being mature enough to handle things in my stead. More so for not making me feel too bad that I hadn't been physically there for him to watch him grow up, all while making me immensely proud of him.

Finally, many thanks to the anonymous EU/UK taxpayer that through its involuntary contribution made it possible to fund this research. I sincerely hope that one day the findings will affect your everyday life in a meaningful manner.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified. Some of the material used in this thesis has been published in the following paper:

- Georgios Stefanakis, Vijay Nagarajan, and Marcelo Cintra. Understanding the Effects of Data Corruption on Application Behavior Based on Data Characteristics, In *Proceeding of the 34th International Conference on Computer Safety, Reliability & Security (SAFECOMP)*, September 2015.

(*Georgios Stefanakis*)

Table of Contents

List of Figures xi

List of Tables xiii

1 Introduction 1

1.1 Fault Masking 3

1.2 Problem Statement 5

1.3 Thesis Goals and Approach 7

1.4 Thesis Contributions 9

1.5 Thesis Overview 10

2 Characterization Framework 13

2.1 Framework Overview 14

2.2 Single-Fault Injection Tool 15

2.2.1 Fault Trigger 16

2.2.2 Fault Injection and Fault Model 16

2.2.3 Monitoring, Data Tracking and Reporting 18

2.2.4 Benefits of Binary Instrumentation 22

2.3 Summary 23

3 Application Behavior Characterization Under Data Corruption 25

3.1 Experimental Setup 26

3.2 Workload-Related Vulnerability Variation 30

3.3 Application Data Vulnerability Variation 31

3.3.1 Per-Bit Vulnerability Variation within Application Data 34

3.4 Memory Space Vulnerability Variation 40

3.5 Register File Vulnerability Variation 43

3.6 Instruction-Level Vulnerability Variation 47

3.6.1 Per Instruction Type at Fault Injection 48
3.6.2 Per Instruction Type at First Consumption of Corrupted Operand 52
3.6.3 Program Space Vulnerability 56
3.6.4 Program Vulnerability Phases 59
3.7 Summary 62

4 Exploiting Application Behavior Characterization 65
4.1 Exploitation Alternatives 66
4.1.1 Identifying Areas of High Exploitability Potential 72
4.2 Data Cache Content Profiling 75
4.2.1 Data Cache Content Profiler 75
4.2.2 Experimental Setup and Profiling Results 79
4.3 Per-Bit Data-Level Vulnerability Exploitation Benefits in a Data Cache 83
4.3.1 Strongly-Protected Surface Trade-Offs Against Fault Coverage 86
4.3.2 Practicality Issues of This Case Study 94
4.4 Summary 96

5 Related Work 99
5.1 Monitoring Behavior Under Data Corruption 101
5.1.1 Analytical Techniques 101
5.1.2 Experimental Techniques - Fault Injection 102
5.2 Observing Vulnerability Variations 104
5.3 Shifting to a Hardware/Software Co-Approach 107
5.4 Fault-Tolerant Cache Design 110
5.5 Summary 115

6 Conclusion 117
6.1 Summary 118
6.2 Contributions 120
6.3 Future Works 121

A Single-Fault Injection Tool Output Specification 125

Bibliography 131

List of Figures

1.1 Visualization of fault-masking effects and corruption outcomes 5

2.1 Complete SWIFI framework overview 15

2.2 Decision tree for determining application data usage type 18

3.1 Workload-related vulnerability variation (Full NPB) 31

3.2 Data memory location related vulnerability variation (Full NPB) . . . 32

3.3 Data size related vulnerability variation (Full NPB) 32

3.4 Data user related vulnerability variation (Full NPB) 33

3.5 Data usage type related vulnerability variation (Full NPB) 33

3.6 Per-bit vulnerability variation (Full NPB) 35

3.7 Per-bit vulnerability variation of FP 8 byte data (Full NPB) 36

3.8 Per-bit vulnerability variation of INT 8 byte data (Full NPB) 36

3.9 Per-bit vulnerability variation of IP 8 byte data (Full NPB) 37

3.10 Per-bit vulnerability variation of PTR 8 byte data (Full NPB) 37

3.11 Per-bit vulnerability variation of PTRMR 8 byte data (Full NPB) . . . 38

3.12 Per-bit vulnerability variation of PTRTP 8 byte data (Full NPB) . . . 38

3.13 Per-bit vulnerability variation of PTRTP 4 byte data (Full NPB) . . . 39

3.14 Per-bit vulnerability variation of INT 4 byte data (Full NPB) 39

3.15 Memory space vulnerability variation (BT, CG, DC, EP, FT) 41

3.16 Memory space vulnerability variation (IS, LU, MG, SP, UA) 42

3.17 Register file vulnerability variation (Full NPB) 44

3.18 Register file vulnerability variation (BT, CG, DC, EP, FT) 45

3.19 Register file vulnerability variation (IS, LU, MG, SP, UA) 46

3.20 Instruction vulnerability variation at fault injection (Full NPB) 48

3.21 Instruction vulnerability variation at fault injection (BT, CG, DC, EP, FT) 49

3.22 Instruction vulnerability variation at fault injection (IS, LU, MG, SP, UA) 50
3.23 Instruction vulnerability variation at first consumption of corrupted operand (Full NPB) 52
3.24 Instruction vulnerability variation at first consumption of corrupted operand (BT, CG, DC, EP, FT) 53
3.25 Instruction vulnerability variation at fault consumption of corrupted operand (IS, LU, MG, SP, UA) 54
3.26 Program space vulnerability variation (BT, CG, DC, EP, FT) 57
3.27 Program space vulnerability variation (IS, LU, MG, SP, UA) 58
3.28 Program vulnerability phases (BT, CG, DC, EP, FT) 60
3.29 Program vulnerability phases (IS, LU, MG, SP, UA) 61

4.1 Visualization of the exploitation alternatives 67
4.2 Management of concurrent usage type classification for in-order data cache content profiling 78
4.3 Data cache content profiling per usage type (over execution time) (CG) 81
4.4 Data cache content profiling per usage type (averaged) (Full NPB) . . 82
4.5 Trading-off fault coverage of FP 8 byte data (Full NPB) 89
4.6 Trading-off fault coverage of INT 8 byte data (Full NPB) 90
4.7 Trading-off fault coverage of PTR 8 byte data (Full NPB) 90
4.8 Trading-off fault coverage of PTRTP 8 byte data (Full NPB) 91
4.9 Trading-off fault coverage of PTRTP 4 byte data (Full NPB) 92

List of Tables

2.1 Reported corruption characteristics 20
2.2 Reported corruption effects 21

3.1 Profiling and sampling information for each tested benchmark 29

4.1 Breakdown of data cache contents per data type and data size (64K cache) (Full NPB) 85
4.2 'Unused' data cache contents/space (64K cache) (Full NPB) 93

Chapter 1

Introduction

Hardware reliability challenges have always been present in all parts of a system (logic, interconnects, memory elements). Examples include wire-induced noise corrupting data communicating over the interconnects, operating voltage/frequency scaling causing timing violations, gamma ray and alpha particle strikes flipping values in transistors, ageing or manufacturing defects causing deterioration of chips and systems.

All such occurrences of anomalous physical conditions are called *hardware faults* [45]. According to their duration, hardware faults can be classified into (a) *transient faults* (soft errors or single event upsets) that occur only once and do not persist, (b) *permanent faults* (hard faults) that occur at some point in time and persist from that time onward and (c) *intermittent faults* that occur repeatedly but not continuously in the same location.

Once a hardware fault occurs, it is upsetting logic values. Therefore, regardless of the cause or the duration of hardware faults, all of them essentially lead to data corruption. As data corruptions affect a system's dependability, hardware faults must be dealt with to ensure the hardware's reliability.

Hardware faults are not a recent challenge and extensive research work is already present to enable fault-tolerant features in a system. The purpose of *fault-tolerant design* is to improve dependability by enabling a system to perform its intended behavior in the presence of a given number of faults [32].

As such, hardware faults need to be detected and corrected to ensure the correct operation of a system. The approach to tolerate a fault depends on its type. Generally, a set of up to four steps needs to be followed in order to tolerate a fault [45]: (1) *Error detection*, where the system becomes aware of the presence of a fault, (2) *Error*

recovery (error correction), where the system tries to mask the fault's effect and set the correct expected behavior, (3) *Fault diagnosis*, where the type and location of the fault is identified (not for transient faults), and (4) *Self repair*, where the system is repaired or reconfigured to avoid the diagnosed fault.

Each step requires a form of redundancy in terms of extra hardware and/or computations. Thus, reliability solutions always come with associated performance and cost overheads in their effort to ensure the highest possible fault coverage.

As it is desirable for systems to maintain their expected functionality, even in the presence of faults, hardware reliability and correctness has become a design target equally important to high performance and low cost. Due to the necessity of a correctly operating infrastructure, especially on safety and mission critical systems, a common approach has been to deal with hardware faults exclusively at the hardware level. In that effort to ensure that the hardware operates correctly, all system components are protected equally, exhaustively and transparently between the hardware/software stack levels. Not unexpectedly, adding the necessary redundancy comes with an overhead of extra operating costs and/or reduced performance. Such reliability-associated costs are not an issue for the safety and mission critical systems.

As we move into deep submicron fabrication technologies the reliability challenges start to appear in an ever-increasing rate [40]. As a result reliability concerns now arise in commodity systems too. Once more the general consensus is to protect all system components equally, exhaustively and transparently between the hardware/software levels. This time though, due to the cost concerns in commodity systems, the various proposed reliability mechanisms aim to maximize dependability under a minimum performance/cost penalty. As such, most existing reliability approaches focus on reducing these reliability associated overheads by tinkering with the implementation details. All these while ensuring that the hardware faults are hidden from the upper architectural/software layers in order to still provide them with the behavior of a fault-free hardware layer.

Unfortunately, due to the increasing error rates in successive process generations, this black box approach cannot be maintained for much longer [9]. We are reaching the point where it will be unsustainable to deal with all the faults at the hardware level transparently. It will become prohibitively expensive, both in terms of power/area cost and performance to tolerate all hardware faults and guarantee a fully protected underlying hardware.

All these lead to the general problem that motivated this thesis; *increasing error rates inducing unsustainable higher reliability costs, especially due to the equal and exhaustive protection.*

The rest of this introductory chapter is organized as follows: Section 1.1 introduces the natural fault-masking effects that are the motivation to our proposed solution to the problem of unsustainably increasing reliability costs. Section 1.2 shifts the original problem into the two new problems that this thesis intends to tackle in order to reduce the reliability costs. Section 1.3 states the goals of this thesis and overviews our approach to achieve them. Section 1.4 states our main contributions. Finally, Section 1.5 concludes this chapter with an overview of this thesis.

1.1 Fault Masking

Interestingly, not all hardware faults end up manifesting as errors in the system's behavior and the executing application's behavior.

In fact, depending on the fault characteristics (location, type, timing, duration), the executing workload and the underlying hardware, a hardware fault may get masked for a variety of reasons by various levels of fault-masking effects [45]:

- *Logic-level masking*, where a hardware fault may get masked at logic gate level before becoming an error, e.g., a 2-input AND gate's output will not change if it has a fault in one input and a zero in its other input.

- *Architecture-level masking*, where a hardware fault is not logically masked but may never propagate to the architectural state, e.g., a hardware fault in memory in the non-opcode bits of a NOP instruction or a hardware fault in the branch predictor.

- *Application-level masking*, where a hardware fault is not architecturally masked but may never propagate to the application's output, e.g., a hardware fault in a memory location that is never to be accessed again.

Once a fault does get masked by any fault-masking level, then the system and the executing application behave as expected resulting into a *correct application execution*. If a hardware fault does not get masked and becomes an error, the effect on the executing application varies and may be one of the following:

- *Application crash*, where the not-masked hardware fault leads to an invalid operation that causes the application to crash.
- *Application stall*, where the not-masked hardware fault causes the application to wrongly wait for an event that will never occur or lead the application to enter an infinite execution.
- *Delayed correct execution*, where the not-masked hardware fault does not corrupt the application output but causes the computation to complete in a larger-than-usual delay.
- *Silent data corruption (SDC)*, where the not-masked hardware fault propagates to the application output causing it to be different than the expected one. Contrary to all the previous execution upsets that are observable when they occur, these cases where the output is wrong are unobservable execution upsets. They can be identified *only* after application completion by comparing the effective application output to the expected correct one. This non-observability of SDCs make them the most severe case of possible execution outcomes under hardware faults. There is also a special case of SDCs, that will not be considered by this thesis, where the output corruption is masked at user level by not being perceived as a wrong output, e.g., a single wrong pixel color in a frame of a video.

Fig. 1.1 visually depicts the possible fault-masking levels and the possible corrupted execution outcomes. To summarize, depending on the fault characteristics (location, type, timing, duration), the executing workload and the underlying hardware, faults can either (a) get masked by various levels of fault-masking effects (logic-, architectural-, application-level) and result in a correct execution with no visible effects or (b) not get masked and result either in an observable execution upset (application crash, stall or delay) or an unobservable output corruption (silent data corruption).

All these fault-masking effects have been widely documented by prior works, especially those that experimentally evaluate the dependability of fault-tolerant approaches. In such works a breakdown of possible execution outcomes is usually presented and the fault coverage is commonly measured as the percentage of the resulting SDCs.

Generally, the focus has always been on reducing the occurrence of SDCs because they are the most severe ones. SDCs are the only execution outcome that is undetectable online by any means and that does not provide any indication that something happened out of the ordinary. All other corruption outcomes are observable and de-

tectable by conventional methods, e.g., software-visible symptom-based fault detection [50, 27, 37, 13, 8] monitors for suspicious behaviors/symptoms, such as violating likely program invariants, memory exceptions, cache misses, branch mispredictions, fatal exceptions, program crashes, high OS activity, hangs.

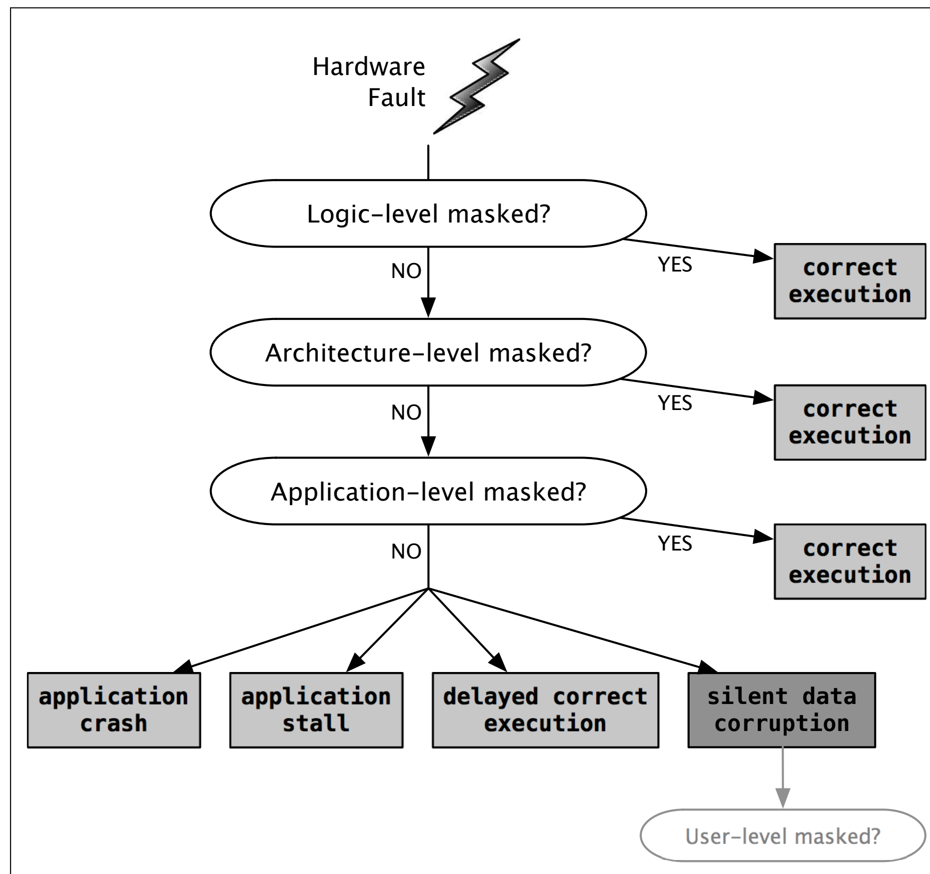

Figure 1.1: A hardware fault possibly getting masked by different levels of fault-masking effects and resulting into different possible corruption outcomes.

1.2 Problem Statement

As mentioned, reliability solutions face the problem of the unsustainably increasing overhead costs as the hardware fault error rates increase, especially due to the tendency to protect all hardware and software components equally and exhaustively.

Mainly motivated by the aforementioned fact that not all hardware faults manifest as the same symptom in an application's execution and can be masked by various levels of fault-masking effects, this thesis proposes a solution to the unsustainable reliability

costs. We propose that we can avoid protecting equally and exhaustively the underlying hardware. Instead we suggest a shift to vulnerability-driven unequal-protection mechanisms, where the protection strength is assigned according to the error sensitivity of the components under protection. Our proposed shift to unequal protection will be targeting a given structure or same-level structures.

Given that the end-to-end effect of a hardware fault on the application output is what matters and, in particular, since only faults that lead to SDCs are what need to be protected against, the area that needs to be protected strongly against faults can be reduced. This intuitively translates into reliability overhead reductions without affecting the fault coverage. Alternatively, this points out that for the same overhead the fault coverage can be increased and help sustain the increasing error rates. This is contrary to most of existing fault-tolerance approaches that tend to be conservative and do not leverage the inherent masking of hardware faults.

As what we propose is to assign protection to hardware/software components according to their likelihood that a fault occurring there will cause an SDC to the final execution outcome, this thesis intends to tackle the problem of driving and implementing vulnerability-driven reliability mechanisms.

This means that in order to solve the original problem of high unsustainable reliability costs we shift the problem into:

- observing and characterizing the exact end-to-end effects of hardware faults on application behavior, and

- exploiting the gained insight in a vulnerability-driven reliability mechanism to reduce the reliability overheads in a given structure or same-level structures.

Generally we expect that useful insight can be gained by characterizing the application behavior under hardware-induced data corruption conditions. In particular, we expect that application behavior under unreliable conditions will vary and we intend to exploit these variations to our benefit. Given the expected error sensitivity variations, we will use this to avoid exhaustive equal protection and reduce the reliability overheads, without a significant reliability impact, by shifting to a class of vulnerability-aware unequal-protection architectures, where less-vulnerable application data are protected less against corruption than their more-vulnerable counterparts.

1.3 Thesis Goals and Approach

Our proposed thesis goals revolve around investigating how the effect of hardware-induced data corruptions on application behavior varies with the purpose of identifying exploitable insight in order to reduce the reliability costs by shifting into a vulnerability-aware protection paradigm.

The goals of this thesis relate exactly to the twofold problem we intend to tackle. Generally speaking, the research goal of this thesis is to answer the following questions: (1) *Can application behavior under data corruption be characterized based on the characteristics of the corrupted data (type, bit, location, etc.) and what data-level error-sensitivity observations can be made?* (2) *How can this characterization insight be exploited to drive the design of lower-cost vulnerability-aware unequally-protected architectures?*

To **characterize the effects of hardware-induced data corruption on application behavior based on the corrupted data characteristics and the executing workload**, we use software-implemented fault injection (SWIFI). This is to model transient fault data corruption in memory locations, during the operation of an application in a unprotected system, and to capture the effect of the corruption on the execution.

As our focus is on gaining detailed error-sensitivity insight of application data types, we develop a portable instrumentation-based SWIFI tool that operates at *application data level*. Given an application binary, without need of its source code, the SWIFI tool can finely control the location of the corruption in the application's memory address space without further intruding the application behavior. Once a fault is injected at runtime, without need for binary file modifications per injection test, it tracks the corrupted data to classify them according to their use by the application. Meanwhile it monitors the execution's state and outcome to report back many diagnostics regarding the corruption and the corruption outcome. Since we monitor the full application behavior under data corruption until completion, we capture all higher levels of fault-masking effects and all possible outcomes of a data corruption (ranging from silent data corruption on one side of the spectrum to correct execution with no visible effects on the other side).

SWIFI is an established method commonly used in the literature as it is a straightforward, fast and easy to deploy way to inject realistic faults in real systems, while allowing for more detailed monitoring/reporting. It has been used mainly for system-

level dependability assessment of reliability mechanisms [20, 42, 48, 7, 39] but usually without further elaboration on the vulnerabilities of individual hardware/software structures. Other non fault-injection approaches do vulnerability estimation through detailed modeling to analyze the behavior of only higher-level hardware structures [31, 46, 47]. Instead, we employ SWIFI in order to gain detailed *error sensitivity insight* at a lower *application data level*.

The main value of our study stems from investigating data-level (down to bit-level) vulnerability variations. As we can perform an extensive set of individual fault-injection tests that report many diagnostic information, we can thoroughly *characterize the data corruption effects on application behavior based on the characteristics of the corrupted data*; namely, among others, their location within the memory address space, their size, their type as used by the application and, even, the exact bit location of the corruption. This enables us to quantitatively investigate how the acuteness of the effect of a hardware-induced data corruption on application varies. As a result we can observe the *actual* varying vulnerability characteristics at application *data-level*, down to identifying bit ranges within specific types of application data that if corrupted by a single-event-upset are less prone to manifest as SDCs. This approach is more detailed compared to existing studies, that also focus on vulnerability estimation, but are limited due to investigating higher abstraction structures (e.g., specific hardware structures [31, 46], code segments [13], instructions [6, 29]) or due to using limited assumptions for estimating the vulnerabilities (e.g., characterizing individual memory locations only by their liveliness attributes [29]). Moreover, as hardware functionality is largely visible through software, the injected faults can emulate faults at various levels of a system and not only at application data level. As such we can extrapolate the insight to error sensitivity of instructions, memory address space and registers too.

After correlating the corruption characteristics to the reported corruption effects on the application behavior, we gain statistical insight into the application data vulnerabilities per tested workload. It is observed that there are vulnerability variations between application data, even within bit ranges of specific application data types.

Given that the gained insight from the characterization study essentially estimates *data-level vulnerability factors* for a given workload, it can be **exploited by shifting to vulnerability-driven unequally-protected architectures** where the application data are protected unequally according to their expected vulnerability. Moving down into characterizing data-level vulnerabilities, instead of higher-level hardware/software

structures, results into deeper and more exploitable insight by a variety of protection schemes.

One promising way to exploit such insight is for reducing the cost of existing fault-tolerant mechanisms that tend to protect everything equally. We argue that such a shift is highly beneficial; the protection overhead can be reduced without a significant impact on fault coverage, as we reduce the fault-protected surface by avoiding the excessive full protection of the previously identified less-vulnerable application data. This is contrary to traditional reliability approaches that either unnecessarily protect all parts equally and exhaustively [16, 19, 52, 11] or offer unequal protection agnostic to the actual distribution of likelihood of the application's data to corrupt the execution output [55, 22, 23].

Achieving these goals will allow to sustain the same reliability QoS as before but for lower operating costs or higher performance. This can be translated also into sustaining a reliable operation of applications on the increasingly unreliable hardware.

Moreover, as our proposed class of vulnerability-aware fault-tolerant architectures can have varying fault protection levels, it lends itself naturally to offer various degrees of *reliability QoS* for trading off reliability against performance/cost. Considering that our characterization study goes down to bit-level vulnerability estimation, the performance/cost benefits can be maximized for an even smoother trading off against reliability. This also enables graceful degrading architectures where for a given error rate we may achieve better metrics than existing techniques and guarantee a minimum set of executing functions.

Finally, vulnerability-aware unequal protection is orthogonal to the specifics of fault-tolerant approaches. It may be applied to any kind of software- or hardware-level fault tolerance method to reduce the total area under protection and, subsequently, their reliability overhead.

As an instance to demonstrate the applicability and potential of exploiting the varying vulnerability characteristics of application data, we assume a data cache design that can unequally protect its contents driven by our characterization insight and we observe the trade-offs between the fault-protected surface reduction against the fault coverage.

1.4 Thesis Contributions

In this thesis, we make the following main contributions:

- We establish an easy-to-deploy instrumentation-based SWIFI framework that

can perform extensive tests on target binaries for a data-aware characterization study of the exact data corruption effects on application behavior, where all possible levels of fault-masking effects are captured.

- After extensive experimental fault-injection tests, we observe different levels of vulnerability variations. The most promising among them is the workload-related vulnerability variation of application data of the NPB-serial benchmarks based on their characteristics, along with the vulnerability variation within parts of application data. For given application data characteristics, we identify clear patterns of less-vulnerable bit ranges that if corrupted are less likely to cause SDCs, e.g., up to 32 LSBs of floating-point data in the CG (Conjugate Gradient) benchmark have each less than 1% probability to cause an SDC. The other observed vulnerability variations are in areas such as the memory space, the register file, individual instructions and the program space.

- We demonstrate the potential exploitability of our data-level characterization findings in a generic fault-tolerant data cache running the NPB-serial benchmarks. Assuming a vulnerability-aware unequal-protection mechanism we show how much we can exploit the vulnerability characteristics of application data and to what effect on the reliability QoS level, e.g., the fault-protected surface of a 64K data cache can be reduced by 41% with a less than 0.01% drop in the fault coverage just by avoiding protection of the less-vulnerable bit ranges of floating-point data in the EP (Embarrassingly Parallel) benchmark.

1.5 Thesis Overview

The remainder of this thesis is structured as follows:

In **Chapter 2** we describe our SWIFI-based framework for enabling our data-aware characterization study of the exact data corruption effects on application behavior. Additional relevant supplementary material is detailed in **Appendix A**.

In **Chapter 3** we present and analyze the characterization results after invoking our SWIFI framework to extensively test the NAS Parallel Benchmarks. The application behavior variation is observed in relation to different levels of corruption characteristics; application data characteristics (usage type, size, location in memory space, bit location corrupted, etc.), memory space characteristics, register-level characteristics and instruction-level characteristics.

In **Chapter 4** we discuss the importance of our characterization findings and on the ways they can be exploited for vulnerability-driven protection of a given structure or same-level structures. Then we demonstrate the potential of shifting to a class of vulnerability-aware unequal-protection architectures by exploiting our insight in a fault-tolerant data cache with two distinct levels of protection strength.

In **Chapter 5** some background information on fault tolerance is discussed followed by relevant prior works. The presented relevant studies generally relate to methods that investigate the varying vulnerabilities of hardware and software components, methods that attempt to unequally protect a system and methods that trade-off reliability against cost/performance.

Finally, **Chapter 6** concludes this thesis by summarizing its approach and contributions, before presenting some grounds for future works.

Chapter 2

Characterization Framework

In this chapter we present our instrumentation-based software-implemented fault-injection (SWIFI) framework that can perform extensive data-level aware fault-injection tests. Its end purpose is to capture the exact effects of hardware-induced data corruption on application behavior, mainly in relation to the corrupted data characteristics and the executing workload.

To do so, we employ SWIFI to model transient single-bit fault in memory locations, during application execution in an unprotected system, and to capture the corruption effect on the execution. As we focus mainly on characterizing the effects of data corruption at data level, we develop a portable instrumentation-based SWIFI tool that is data-level aware. Given an application binary, without need of its source code, our tool can finely control the location of the corruption in the application's memory space without further intruding the application behavior. Once a fault is injected at runtime, without need for binary file modifications per test, on top of traditional fault-injection approaches, it tracks the corrupted data to gather more information regarding them. Meanwhile it keeps monitoring the execution to report back more diagnostics relating to the corruption characteristics and the corruption outcome.

We opted to perform the characterization study through fault injection testing, instead of modeling-based methods. Fault injection is a straightforward technique that allows us to quickly set a toolchain to capture all possible end-to-end effects of data corruption down to bit-level granularity, in real applications in real systems. If we were to use analytical techniques, the propagation of every bit of every used data would have to be modeled through a detailed hardware/software data flow analysis. Then, to obtain the desired vulnerability characteristics, a slow detailed microarchitectural simulation

of the tested application would have to be performed. Despite the time effort, this analytical approach would still not be able to capture all the detailed corruption characteristics and effects, as we intend.

Moreover, due to implementing the SWIFI framework using binary instrumentation in particular, the close monitoring of the application execution status is possible without requiring all of the above. Dynamic instrumentation is an easy to deploy, portable and efficient way to alter and monitor an application's execution. On top of modeling the desired corruption and capturing the exact execution outcome of real applications in real systems, fault injection by instrumentation also allows us to finely control the location of corruption, to closely observe the execution and to capture even finer details of the corruption characteristics and the corruption effects.

In the rest of this chapter, first, we present an overview of our framework in Section 2.1. Then, to show how we can capture all the desired properties of our experiments, in Section 2.2 we elaborate on the details of the individual single-fault injection process, focusing on the chosen injected fault model and the data-tracking capabilities of the implemented tool.

2.1 Framework Overview

Our proposed **SWIFI framework** (Fig. 2.1) operates as follows on a given application-under-test to characterize its behavior under data corruption:

- First, a golden run of the target binary is profiled, without introducing data corruption, to obtain (a) its expected correct output (for SDC detection), (b) its normal execution time when under instrumentation by our tool (for delayed/stalled execution detection) and (c) its total number of memory load accesses (for deciding the sample rate to drive the tests uniformly over the test space).

- Then a *single-fault injection tool* is repeatedly invoked on a clean instance of the target application, each time corrupting a different memory load operation. In every test, the application-under-test is re-executed, a bit-flip is injected at a random bit of a different specified memory load access and the rest of the execution is closely monitored to capture/report the details of the corrupted data characteristics and of the exact corruption effects. Given a sample rate, we drive the tests uniformly over the test space of all possible memory locations and times.

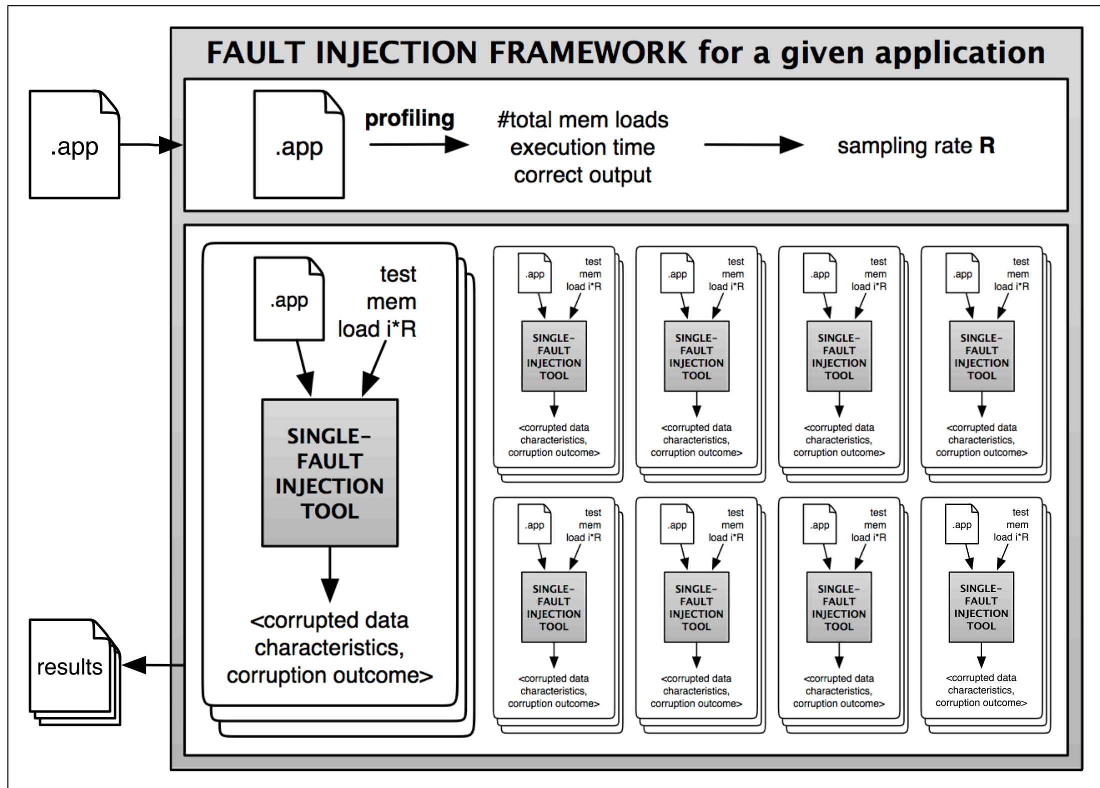

Figure 2.1: Our proposed framework that captures an application's behavior under data corruption through extensive multiple single-fault injection experiments.

- Finally, once all tests complete, the extensive reported execution results are aggregated to relate the corruption outcome to the corrupted data characteristics (Chapter 3).

Once we set up the framework, it will run extensive fault injection tests over given applications. Due to the volume of our experiments and the detailed reported diagnostics, after aggregating our results and studying them, we will be able to confidently relate in many ways the corruption attributes and the resulting corruption effect on the application. Among the gained insight, we expect to be able to elaborate on the varying vulnerability factors of application data depending on their characteristics, down to even identifying different vulnerability levels within bit ranges of application data.

2.2 Single-Fault Injection Tool

At the heart of our framework is the **single-fault injection tool** that operates following the established fault injection methodology. In each test once the **fault trigger** condition is reached, a **fault injection** is performed according to the chosen **fault model**

and the rest of the execution is **monitored** to **report** the exact end-to-end effect of the corruption. Since we opted for SWIFI these operations are performed using special software to emulate the behavior of expected hardware faults during the application operation only (and not the kernel).

To briefly outline the operation of the single-fault injection tool, it performs and monitors the fault injection tests by dynamic binary instrumentation of the application-under-test. In each test just before a specified memory load access a random bit of the accessed data is flipped to emulate a single-bit transient fault in the particular memory location. Then the execution is monitored and, on top of usual SWIFI approaches, the corrupted data are tracked in order to report more detailed corruption characteristics (Table 2.1) and corruption effects (Table 2.2).

2.2.1 Fault Trigger

First, the single-fault injection tool starts by instrumenting the application-under-test until the execution reaches the specified memory load access to be corrupted. The memory load access to be corrupted will be chosen uniformly out of the total memory load accesses of the tested workload.

The specified memory load access acts as both a spatial and a temporal fault trigger to invoke the injection routine just before the load operation. Instead of corrupting random memory locations in the whole address space at random times, using the memory load access as a fault trigger captures all possible times that a transient fault could occur and all possible live memory locations that could get corrupted by a transient fault. Thus it simplifies driving where/when to inject a fault as it narrows down to selecting a memory load access, without having to rely on external events. Also its inject-before-load policy reduces the testing space by avoiding the unnecessary testing of dead memory locations.

2.2.2 Fault Injection and Fault Model

Once the instrumentation code detects that the fault trigger condition has been reached, the fault-injection routine is invoked in a manner similar to a software trap. Then, complying to our fault model, it injects a random bit-flip in the accessed data just before they are accessed.

The chosen injected fault model emulates *single-bit transient faults in memory locations* by randomly flipping a bit of the accessed data just before they are accessed

by a memory load operation. The bit-flip is performed by storing the corrupted value to the same memory location without further intruding the application's original behavior. This is to force the execution to behave as if a different value was already there and avoid activating any reliability mechanisms of the system that would skew our intended vulnerability insight.

Software-injected high-level faults model lower-level hardware-induced data corruption to make the system behave as if a hardware fault was present in order to monitor the application behavior. Therefore, it is important in fault-injection testing for dependability assessment to choose fault models that closely resemble the naturally occurring hardware faults. Given that we focus on error sensitivity of application data, the chosen fault model suffices for our purposes to capture the application behavior under corrupted application data without a need for precise realistic hardware fault models. Plus, due to the instrumentation-based SWIFI the fault model does not need to be adapted per target system but only to have the necessary high-level characteristics (type, duration, location) required by our application data level investigation.

In particular, we chose a bit-flip *fault type* instead of stuck-at-zero or stuck-at-one to ensure that application data will always be corrupted at every injection test.

Moreover, the inject-before-load policy enforces emulation of transient faults as the injected corruption will not persist after the corrupted location is overwritten. Modeling transient *fault duration* fits better our purposes as they affect only a single memory object. On the contrary, permanent/intermittent (or multi-bit) faults in memory locations, besides being harder to emulate in software, corrupt different data over time (or multiple bits). In that case it would be indistinguishable which corrupted data (or bits) are responsible for the reported outcome.

Finally, the injected *fault location* is in main memory and it suffices for our data-aware error sensitivity investigation. Although we inject fault at main memory locations, our chosen fault model is not limited to emulating hardware faults only in memory. Depending on the system and how the fault is propagated, it may translate to emulating faulty behavior in other locations too, such as the register and the cache location where the corrupted value is loaded, or faulty behavior of communication wires and CPU instructions that use the corrupted value.

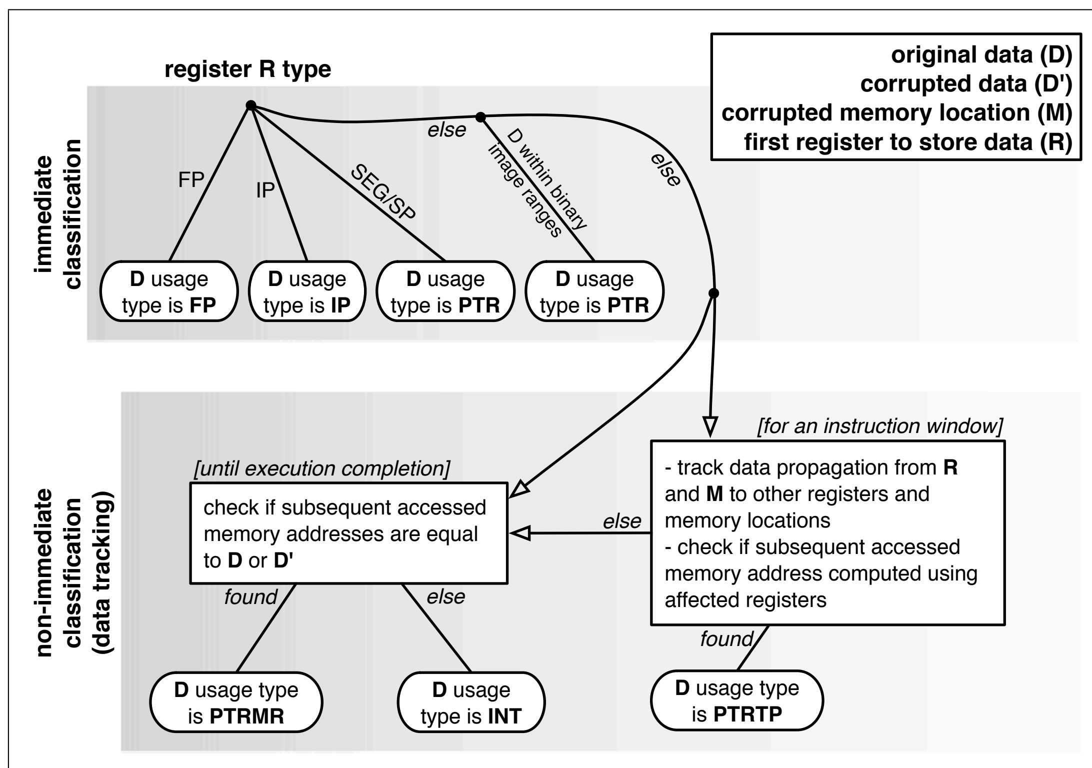

Figure 2.2: Decision tree used by the single-fault injection tool to classify the corrupted application data according to their use by the application.

2.2.3 Monitoring, Data Tracking and Reporting

Once the fault has been injected the rest of the application execution is still instrumented and analyzed to monitor and report the effects of the data corruption. As the instrumentation and analysis are transparent to the original binary behavior, the fault-injection software does not intrude the application space/behavior and guarantees the non-intrusiveness of the SWIFI tool.

Due to the chosen fault trigger, fault model and instrumentation-based injection we can perform data-level aware fault injection and data-aware characterization of corruption effects. Once the memory location is corrupted and then loaded, we can track it as an application variable to get its detailed usage characteristics and relate them to the outcome to capture the varying vulnerabilities of application data.

More precisely, at the moment of the fault injection, the tool tries to finely identify as many characteristics of the corrupted data as possible (Table 2.1). Attributes such as their location in the memory address space (global, heap or stack), size and user (system or user data) can be identified immediately.

Classifying the type of the corrupted data according to their use by the application

(Fig. 2.2) can be either immediate (e.g., floating-point data, instruction pointer, etc.) or it may require tracking the data through the execution until a first meaningful use (e.g., to determine if they are used for addressing memory or not).

To elaborate, at fault injection which always happens just before a memory load operation, we can identify the first register (R) where the corrupted data (D') are stored. If it is a floating-point register, the instruction counter or a segment/stack pointer register, we can classify immediately the corrupted data as floating-point (FP), instruction pointer (IP) or memory addressing data (PTR) respectively. Otherwise, we check if the original uncorrupted data (D) hold a value within the address ranges of the application binary image and, if they do, we identify them as memory addressing data (PTR).

If the data usage type cannot be determined immediately, data tracking and close monitoring of the execution are used to determine their usage. The corrupted (and the uncorrupted) value is checked against all subsequent accessed effective memory addresses to check if the corrupted data are used for memory addressing (PTRMR).

Meanwhile, for a specified instruction window, we use *dynamic taint analysis* [56] to track the data propagation from the first register (R) to hold the corrupted data and from the corrupted memory location (M). We model our problem as a dynamic taint analysis problem where the taint sources are the register R and the bytes in M. After every instruction, we track accordingly the taint propagation to other registers and memory locations.

To do so, for every executed instruction, we obtain a list of the input/output registers and input/output memory locations accessed by the instruction and:

- If the instruction clears an output register (e.g., `sub R1, R1, R1`), then the output register gets untainted in case it was tainted already.
- If the instruction outputs to registers/memory locations:
  - If *any* of the input registers/memory locations is tainted, then *all* output registers/memory locations must get tainted.
  - Otherwise, if *all* of the input registers/memory locations are untainted, then *all* output registers/memory locations must get untainted.
- If the instruction does not output to any register/memory location:
  - No taint propagation for this instruction.

Table 2.1: Reported corruption characteristics

Characteristics of corrupted data	
Injected bit-flip location	
Memory address of corrupted data	
Memory space location	global, heap or stack
Size	1, 2, 4, 8 or 16 bytes
User	System library data or application-space (user) data
Usage type	**FP**: floating-point data (immediate classification)
	IP: instruction pointer (immediate classification)
	PTR: memory addressing data (immediate classification)
	PTRMR: memory addressing data (classification by checking subsequent accessed memory addresses)
	PTRTP: memory addressing data (classification by data tracking until first use as memory addressing data within an instruction window)
	INT: integer data (if none of the above)
Other corruption characteristics	
Corrupted memory load access number	
First register to store the corrupted data	
Corrupted instruction	Instruction pointer, opcode and time at corruption
First use of corrupted data	Instruction pointer, opcode and time at first use of first register to store the corrupted data
If usage type identified as memory addressing data (PTRMR or PTRTP)	
Time until first use as memory addressing data	
Taint propagation of corrupted data	Number (and total cumulative number) of registers and memory bytes where the corrupted data have propagated through until detection as memory addressing data.

This tracking continues until a tainted register (in other words, a register whose contents have been affected by the original corruption) is used for computing a memory address (when the tainted register is used as a base or index register in a memory operation). If this happens within the specified instruction window, then the original corrupted data are reported as memory addressing data (PTRTP). The requirement to perform dynamic taint analysis for a specified instruction window is to avoid taint explosion, where the taint from register R would end up propagating to all registers and would cause the false positive reporting of the original corrupted data as PTRTP.

Apart from reaching the end of the tracking instruction window, other reasons that stop the tracking are: (a) an event causing the execution to stop (i.e., the application finishes or crashes), and (b) the detection of the corrupted data as PTRMR in the meantime. Finally, if by the end of the execution the usage type is still unclassified, then the corrupted data are reported as integer data (INT).

Table 2.1 summarizes all the reported corruption characteristics. Most relate to the characteristics of the corrupted data (memory address, corrupted bit location, memory

Table 2.2: Reported corruption effects

Corruption effects	
Total number of executed instructions	
Execution outcome	**Correct output**
	Delayed correct output when the total execution time is a set amount of times more than the normal uncorrupted execution time
	Application crash reported along with the crash details, the instruction's opcode that caused the crash and the time in instructions from the corruption until the crash
	Application stall due to excessive total executed instructions (or excessive total execution time), as dictated by specified stall-time ratios, indicating that the corruption caused an infinite execution (or the application to wait)
	Silent data corruption (SDC) wrong output

space location, size, user and usage type). Moreover, due to the close monitoring and data tracking we can take note of other corruption details, such as the first register to store the corrupted data, the first instruction to use the corrupted data, the time until first use of the corrupted data, the time until first use as memory addressing data, how much the corruption propagated in the register file and memory (taint propagation spread in terms of tainted registers and memory bytes), etc. It will later be shown how such information can be used to extrapolate the application data error sensitivity insight to other functional units' error sensitivity (such as the register file) or to instruction-level error sensitivity.

After the injection, apart from possibly tracking the corrupted data to determine their usage type, the tool keeps monitoring closely the execution to capture all possible exact corruption effects (Table 2.2). The execution time is monitored to detect application stalls or delayed executions, the output is checked for correctness or SDCs and fatal signals are caught to identify crashes.

Once the execution completes (or stops due to a crash or a detected stall), the tool reports back all the captured details relating to the corruption characteristics (Table 2.1) and to the observed corruption effects (Table 2.2) for the performed single-fault injection test.[1]

[1]For details regarding the format and the meaning of the reported information, along with more details on the way the reported information was captured, see Appendix A.

2.2.4 Benefits of Binary Instrumentation

It has been mentioned why fault injection is more fitting to our purposes, compared to modeling or simulation approaches, as it has the benefit of being a fast way to capture the exact end-to-end effects of data-corruption.

Moreover the ease of performing fault-injection experiments is further ensured by opting for software-implemented fault-injection (SWIFI), instead of hardware-implemented, that on top of that is a dynamic binary instrumentation based implementation. SWIFI, compared to hardware-implemented fault injection, is more flexible, easier to operate, control and monitor the execution, while being more portable and not requiring special purpose hardware.

Especially due to the use of dynamic binary instrumentation for performing the SWIFI tests, our characterization framework inherits all the benefits of using instrumentation. Dynamic instrumentation is an easy to deploy, portable and efficient way to alter and closely monitor an application's execution. Furthermore, it helps to implement all our desired properties of the experiments; it assists our intended data tracking and detailed execution monitoring, it allows to finely control the corruption location, to closely observe the execution and to capture the intended finer details of the corruption characteristics and the corruption effects (as shown in Tables 2.1-2.2).

All these are achieved without intruding the original behavior of the tested application. As the instrumentation code is contained from the original binary, the injection and monitoring routines are transparent to the executing target application and do not interact with it, apart from injecting the corruption, and do not interfere with the observed application behavior under corruption.

Moreover, as dynamic binary instrumentation allows for a runtime injection implementation, any application can be tested without need for its source code; as long as a native executable binary file is available, it can be fault injected. There is no need for recompilation of the source code or for pre-runtime modification of the original binary to accommodate each different test. The injection can be performed without requiring special architectural features or special hardware to trigger the injection routine or test-specific software/hardware traps.

As the target application does not require modification, many tests can be performed without changing the binary to support specific injections every time. As this enables the automatic injection in the test space, with no manual effort required, a larger number of tests can be performed automatically at a lower effort cost.

2.3 Summary

In this chapter, we presented our instrumentation-based software-implemented fault-injection (SWIFI) framework. It utilizes a single-fault injection tool that is repeatedly invoked to inject single-bit transient faults at uniformly chosen memory load accesses during separate runs of the application-under-test. Its purpose is to monitor the application behavior under data corruption, while tracking the corrupted application data, to report detailed diagnostics regarding the corruption characteristics and the corruption effects.

To summarize the key attributes of our approach, contrary to existing approaches, our framework enables to thoroughly quantify the varying effects of hardware-induced data corruption on application behavior. This is by observing the results of *extensive experiments* (to uniformly test data among the testing space and to confidently elaborate on the relation between the corruption location attributes and its effect) that are *fault-injection-based* (to capture all levels of fault-masking effects and all possible end-to-end exact corruption effects). Moreover, the experiments are *finely controlled* (to control the exact location of the injected corruption), *closely monitored* (to report a variety of detailed diagnostic information regarding the corrupted data characteristics and the corruption effect) and *data-level aware* (to enable our intended novel finer-grained characterization of data corruption effects).

In the next chapter, we employ our framework to extensively test a set of benchmark applications and then aggregate all the detailed reported results in order to characterize their behavior under data corruption.

Chapter 3

Application Behavior Characterization Under Data Corruption

In this chapter, first we describe how we setup our experimental framework to repeatedly test a set of benchmark applications (NPB-serial). Then, after performing our extensive tests and aggregating all the detailed reported results, we discuss our findings towards characterizing the exact effects of data corruption on application behavior based on the attributes of the corruption.

As we will observe how application behavior varies under data corruption, we will be able to elaborate on the varying vulnerability characteristics of application data depending on their characteristics and the executing workload. What we consider as vulnerability of application data is the probability of resulting into a silent data corruption (SDC) if corrupted during the execution of the application.

Although our fault-injection tests corrupt memory location data specifically, our characterization analysis is not limited to investigating application data vulnerabilities only. Due to the nature of the reported corruption characteristics, we can investigate vulnerabilities of other hardware/software areas. This is contrary to existing studies that were limited to estimating the vulnerabilities of only the higher specific abstraction structures under consideration (e.g., specific hardware structures [31, 46], code segments [13], instructions [6, 29]).

In particular, first, the workload-related vulnerability variation is observed. Then it is followed by the data-related variation based on data level characteristics, such as corrupted data type, size, bit location corrupted, etc. Finally, using the remaining corruption characteristics being reported by our framework, some other interesting

vulnerability variations are discussed to elaborate on the error sensitivity of other areas, such as the memory space, the register file, individual instructions, the program space, etc.

Generally, our purpose is to gain exploitable insight and to potentially identify areas with lower vulnerabilities. As we propose that the error sensitivity insight can be used to drive the design of lower-cost vulnerability-aware reliability mechanisms, where protection is assigned according to the protected part's vulnerability to cause an SDC, we will focus on identifying areas with lower probability to result into an SDC.

3.1 Experimental Setup

The proposed SWIFI framework was implemented as a set of scripts and dynamic binary instrumentation Pin tools [28]. The full set of the ten workloads of the NAS Parallel Benchmarks [10] (64-bit, NPB-serial[1] version 3.3.1, input class size S, gcc 4.4.6 -o3, Linux kernel 2.6.32) was extensively tested by our framework on a x86-64 computer cluster [33].

As it usually holds for all experimental evaluation setups, our results too are conditional to the chosen experimental setting. All attributes of the hardware/software configuration (e.g. chosen benchmarks, input sets/size, compiler parameters, hardware parameters) affect the observations to be made. In this instance, the vulnerability variations will be observed for the aforementioned setup of benchmarks, input size and hardware architecture. Nevertheless, here we explain the reasoning behind our choices and how these could affect the observed results.

For the purpose of our vulnerability investigation we tested the NPB applications. The **NAS Parallel Benchmarks** (NPB) are a small set of programs designed to help evaluate the performance of parallel supercomputers [10]. NPB perform highly-iterative scientific computations over test input data generated within each benchmark run. Example computations include fast Fourier transformation (FT), integer sorting (IS), eigenvalue estimation (CG), Heat equation solving (UA), etc. NPB support different input class sizes and have generally little I/O. NPB are not strongly dependent to the generated input resulting into similar execution times for different same-size inputs.

One of the main reasons behind testing the NPB applications was their data-intensive

[1] We tested only the serial versions of NPB as our fault injection tool does not support parallel applications.

scientific nature. As the main focus of this study was to identify the vulnerabilities of application data, such workloads were a good fit to focus on testing the vulnerabilities of data types. One other inadvertent benefit of testing the NPB applications was their reliance on floating-point arithmetic. Due to their FP-heavy nature, FP data were extensively tested and showed similar behavior across the different programs. This established that this observed behavior is fundamental and should hold for other configurations of floating-point intensive codes.

On the other hand the selected benchmarks posed some limitations exactly due to their nature. As they heavily rely on iterative computations, there are more opportunities for faults to get masked by application-level masking effects making the benchmarks inherently more resilient. Nevertheless, the intention of our study has been to capture all masking effects. As a side effect of the iterative nature of the tested programs, a more intensive testing of the more-frequent data became possible and that further increased the confidence of the results to solidify the observed behavior. This held more true especially in regards with the data types that showed consistent behavior across benchmarks.

Finally, another initial reason behind testing the NPB applications was their parallel nature that would enable also to test the vulnerability of data controlling the parallel execution. Alas, only the serial versions of NPB were tested as supporting parallel applications would require significant modifications of the fault injection tool. As expected, this limits the gained insight by not reporting an extra type of application data, those that control the parallel execution. Besides that, the rest insight is expected to hold irrespective of the parallelism or not of the tested application.

Another parameter that affects the results is the input class size and the input data values. There are different input class sizes that can be chosen to execute the NPB applications. We would not expect larger input sizes or different inputs to have a significantly changed vulnerability behavior for a given program, for the case of the tested applications. The NPB programs are highly iterative and would behave similarly. A larger input class would only increase the test space and increase the time to perform the experiments.

Therefore, we opted for the smallest input class size (input class size S). This not only reduced the execution time per benchmark but also the size of the total test space. Although this resulted into less tests performed for a given sampling rate, the number of total tests performed was still significant enough. Moreover, due to the uniform distribution of the tests, memory locations may be tested multiple times over different

runs increasing the confidence of our results. As before, we wouldn't expect larger input sizes to have a significantly changed vulnerability behavior for a given program of the tested applications, especially due to the dependance of the observed vulnerability variations mostly on the type of the tested data.

Similarly, we don't expect significant variations for different input values. As NPB target the performance evaluation of parallel systems, the generated inputs try to generate an equal workload. For that reason, although we ensured that the generated input was the same in every different run of the benchmarks, we expect that the observed vulnerability patterns will still hold for other input values. More so because our most interesting observations are explained due to the nature of the corrupted data types. What we expect would differ in relation to the input values is the exact intensity of the observed vulnerability levels.

To conclude, the observed vulnerability variations depend on the chosen experimental setup. Despite the limitations, we expect that our main observations, that will be discussed throughout the rest of this chapter, can still hold for different setups. As it will be shown later in this chapter, our two main observations relate to (a) the per-bit data level vulnerability behavior of application data and (b) the potential of a per-application vulnerability characterization to generate useful insight at different observation levels. Although the exact results are conditional to this setting and all decisions made would affect the exact observed vulnerability variations, these choices though do not invalidate our observations. Both observations will still hold for different setups because the former (a) reflects a fundamental behavior of application data types and because the later (b) inherently asks for a characterization testing under a given experimental setup before obtaining/exploiting the results.

Before commencing with the individual fault injection tests, the benchmarks were profiled (Table 3.1) to obtain their total memory load accesses and their normal uncorrupted execution time under instrumentation. As the number of total memory load accesses captures all possible memory locations and times to inject a fault (not including all the possible different bits where the fault could be injected within a memory location), it indicates the size of our testing space. Given that in the profiled benchmarks the total memory loads ranged from 4.7 million to 914.9 million, summing up to a cumulative total of 2.28 billion, testing every bit of every memory load access would be impractical and unreasonable. Nevertheless, such an exhaustive testing would provide with the highest possible degree of confidence at the extreme cost of testing time.

Table 3.1: Profiling and sampling information for each tested benchmark

NPB-SERIAL 3.3.1, INPUT CLASS S					
Benchmark	Total memory loads (M)	Execution time (sec)	Sample rate	Memory loads tested (K)	Test space coverage (%)
BT	187.5	20.7	1/234	801.6	0.43
CG	111.9	22.7	1/153	731.3	0.65
DC	33.1	21.9	1/43	769.9	2.33
EP	778.2	52.3	1/2461	316.2	0.04
FT	112.0	31.9	1/216	518.5	0.46
IS	4.7	3.4	1/5	959.3	20.00
LU	62.9	16.5	1/62	1015.6	1.61
MG	10.6	13.7	1/8	1335.0	12.50
SP	66.9	15.3	1/62	1079.3	1.61
UA	914.9	53.3	1/2947	310.4	0.03
Total	2283.1	-	-	7837.6	0.34

Instead we set sample rates per benchmark to uniformly distribute our fault injection tests over the test space of possible memory load accesses to corrupt. To further ensure the uniform distribution of the tested injected corruptions, in every test the bit-flip was randomly injected within the tested data to ensure an equal probability of testing every bit.

An implication of introducing the sample rate, that is using the memory load access to uniformly distribute the tests over the test space, is that more-frequently accessed data may be tested more times than less-frequently accessed data. Given that each test is independent and in every test a different bit is corrupted, this works to our benefit to further ensure the confidence of our vulnerability results when considering a specific combination of application data types. On the other hand, one limitation of sampling based on load accesses is that some accessed data may never be tested. This would not affect our final observation as these are data that are likely not to stay live for long execution periods and thus are less likely to get corrupted and affect the execution. In any case though, for safety reasons, they can still be assumed as critical data even if they haven't be tested.

Table 3.1 shows the chosen sample rates that ranged from 1/5 to 1/2947 to uniformly test each benchmark. They were decided based on the number of memory

loads and normal execution time per benchmark, so that each benchmark is tested in approximately the same total time on the available computer cluster, where the embarrassingly parallel nature of the tests was exploited for a faster completion of the experiments (less than a week). The test space sampling brought the total number of performed fault injection tests to 7.8 million for the full benchmark suite (ranging from 310.4 thousand to 1.33 million for individual benchmarks). Despite the test space sampling, compared to related fault injection based works, we performed significantly more extensive fault injection tests that, coupled with the detailed collected test results, enabled us to thoroughly elaborate on them, as we discuss in the rest of this chapter.

3.2 Workload-Related Vulnerability Variation

In the rest of this chapter an analysis is performed on the gathered information from the injection experiments. Before delving into more exploitable insight, first an analysis of the results is shown at a higher level mainly to show the overall workload-related vulnerability variations.

Fig. 3.1 shows the breakdown of the exact end-to-end corruption effects on the tested benchmarks. Similar to existing studies, the presented breakdown reconfirms that hardware faults have varying effects on application behavior. Out of 7.8 million performed fault injection tests on NPB-serial, 61.1% resulted in correct execution, indicating that exhaustive protection is unnecessary. As for the rest outcomes, 23.5% of the total tests resulted in SDCs, 15% in application crashes, 0.3% in application stalls and less than 0.1% in delayed correct executions.

More importantly it can be observed that the number of faults that corrupted silently the output varied per benchmark; the reported occurrences of SDCs ranged from 5.8% for DC up to 37.9% for IS. Given that it suffices to protect only against SDCs, we can further reduce the amount of data that need strong protection. Protecting only against SDCs suffices because they are the only type of corruption outcome that is undetectable online by any means, plus they do not cause any observable indication that something happened out of the ordinary. All other execution corruption outcomes are observable as they upset the execution in a visible way (i.e., application stall, crash or delay) and detectable by conventional methods (e.g., by software-visible symptom-based fault detection that monitors for suspicious software behaviors/symptoms).

The presented breakdown of corruption effects is for the tests performed following our experimental setup and for the set sample rates (as shown in Table 3.1). These

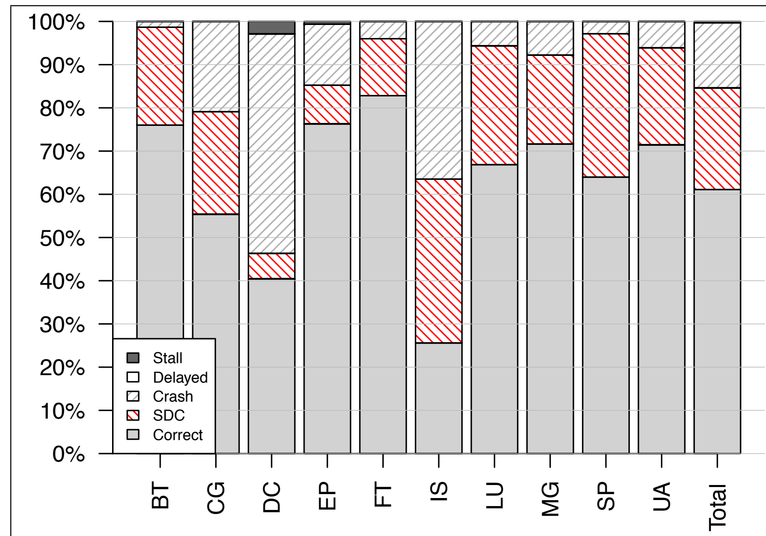

Figure 3.1: Breakdown of application behavior under corruption for each tested NPB-serial application (including total breakdown for all performed tests)

results are cumulative per benchmark and would be different if different sample rates had been chosen. E.g., as the sample rate is driven by the total load accesses, a decreased sample rate would test more the more-frequently accessed data and the total breakdown would be skewed by their vulnerability. Despite that, these results were shown here to demonstrate that there is varying application behavior under data corruption and that this is worth investigating. Therefore, in the rest of this chapter, the vulnerability variations are shown for individual combinations of corrupted location characteristics.

To conclude, Fig. 3.1 showed that application behavior under data corruption varies depending on the benchmark indicating that applications have different inherent vulnerability characteristics. These variations are mostly attributed to each program's data-level sensitivities and their data access patterns. As such, it is promising to move into an application data vulnerability investigation to gain more insight.

3.3 Application Data Vulnerability Variation

Given the volume of our experiments and the detailed captured information of the corrupted data characteristics, we can characterize the application data vulnerabilities based on the data characteristics. For this purpose we introduce an *experiment-based*

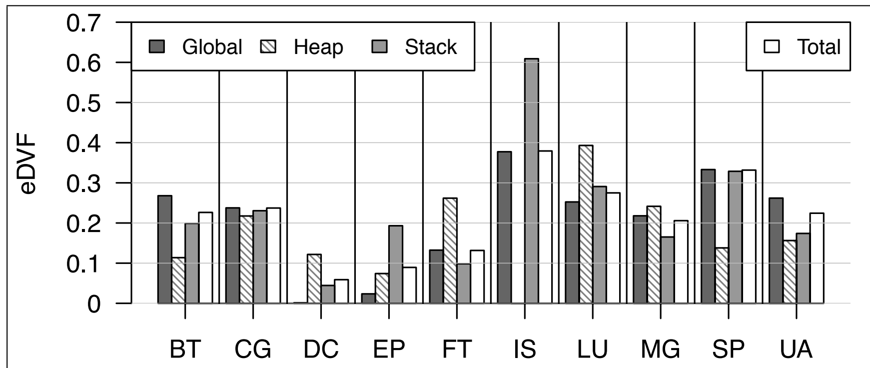

Figure 3.2: Varying ***eDVF*** *depending on the* ***location*** *of the tested data in the memory address space for each NPB-serial benchmark. Total indicates the reported SDCs per benchmark.*

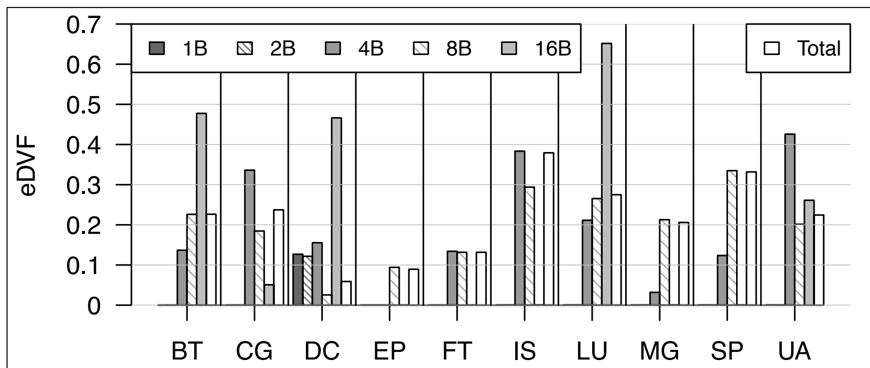

Figure 3.3: Varying ***eDVF*** *depending on the* ***size*** *(in bytes) of the tested data for each NPB-serial benchmark. Total indicates the reported SDCs per benchmark.*

Data Vulnerability Factor (eDVF) that we define as the probability of a corruption in specified data categories to result into an SDC. Contrary to the well-known vulnerability factors that estimate through modeling the probability of a fault in a specific structure for a given application to corrupt the output [31, 46, 47], we use our testing results to calculate the eDVF as the fraction of tests, on application data with specific characteristics, that resulted in an SDC for a given application.

Fig. 3.2-3.5 show the eDVF variation over the tested NPB-serial benchmarks for the different data characteristics that our fault injection framework can identify. Generally most eDVFs are around 0.2, with limited exceptions going as high as 0.83, and quite a few being less than 0.05. This points out that it is possible to rank application data based on their probability to cause an SDC according to the data characteristics. In a few cases, eDVFs are down to zero due to no reported SDC outcomes or due to absence of the particular data categories in the specific benchmark. Both of these causes of zero eDVFs indicate the particular data categories as less-vulnerable for the application in

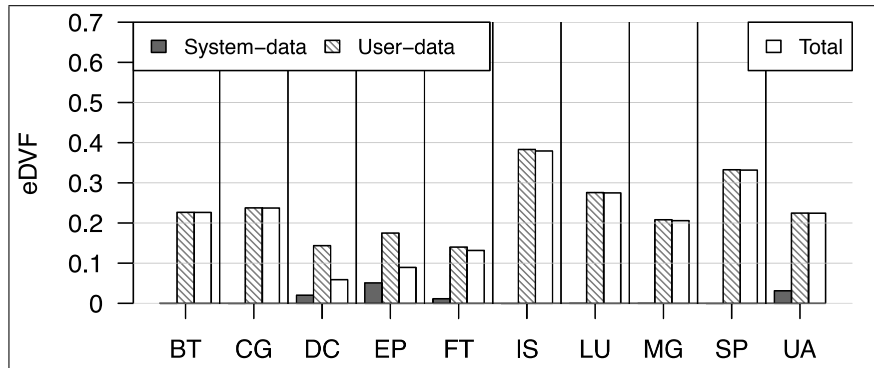

*Figure 3.4: Varying **eDVF** depending on the **user** (system library or user data) of the tested data for each NPB-serial benchmark. Total indicates the reported SDCs per benchmark.*

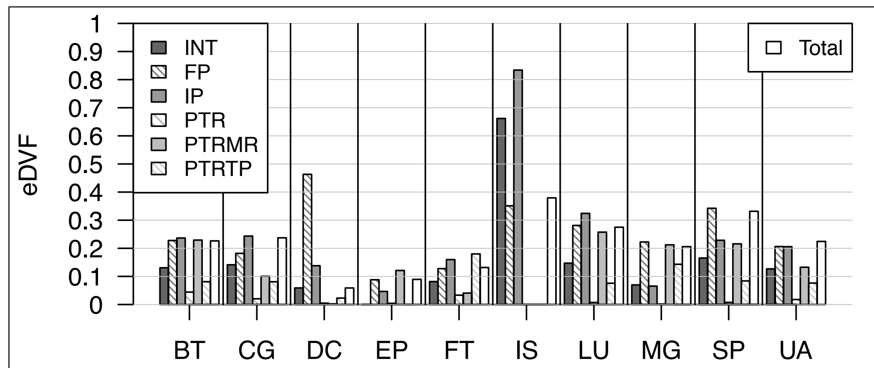

*Figure 3.5: Varying **eDVF** depending on the **usage type** (see Table 2.1) of the tested data for each NPB-serial benchmark. Total indicates the reported SDCs per benchmark.*

question.

System library data (Fig. 3.4) are less vulnerable almost consistently across all benchmarks as they tend not to be output related and if corrupted tend to get masked or cause crashes. On the contrary, in some benchmarks, there is a trend of longer data being more vulnerable (Fig. 3.3), as longer data often hold output-related values and thus if corrupted are more likely to corrupt the output too. This is more evident in the benchmarks that use the longer 16-byte sized `long double` data. As for the usage type eDVFs (Fig. 3.5), there is no benchmark wide observation to be made. Despite that, they can be used per application basis to rank the vulnerability of their application data according to their type. Moreover, given the application's data access patterns, they can explain the total application-level vulnerability variations.

Given the volume of our tests, our study captures the varying vulnerabilities of application data per application with a high statistical confidence. Since there are numerous ways to correlate the application data characteristics to the corruption out-

come, only an informative set per single data characteristics was presented here. The detailed results of our experiments allow us to compute the eDVFs for combined data characteristics to characterize more closely the application data vulnerabilities. Given single-characteristic eDVFs for an arbitrary number of different data characteristics, the combined eDVF for a given benchmark can generally be computed by multiplying all the individual eDVFs with the ratio of the total number of data in the benchmark that fit all the characteristics by the total number of data that fit at least one of the characteristics. E.g., the eDVF of global-8B-user-integers in BT is the product of $eDVF_{BT\text{-}global}$, $eDVF_{BT\text{-}8B}$, $eDVF_{BT\text{-}user}$, $eDVF_{BT\text{-}int}$ and the fraction (number of data in BT that are global-8B-user-int)/(number of data in BT that are either global or 8B or user or int).

Although there seems to be no common behavior of eDVFs across the tested benchmarks, given the volume of our tests, our eDVFs investigation captures the varying vulnerability factors of application data per application. As eDVFs can be computed for all combinations of data characteristics for a given application, they can be exploited to rank application data according to their expected vulnerability to drive a vulnerability-aware protection mechanism, i.e., for a given application we can tell which data types are less prone to cause an SDC and thus protect them less than the rest.

3.3.1 Per-Bit Vulnerability Variation within Application Data

Moving deeper in our investigation we can get more consistent vulnerability insight by observing how the data corruption effects vary depending on the exact bit location of the corruption. As a showcase of this, Fig. 3.6 depicts the reported outcomes of *all* performed fault-injection tests depending on the exact bit location of the injected corruption. As this includes all tests across all benchmarks, it cannot be comprehensively used to estimate the eDVFs per bit for any application data.

Nevertheless it can be seen that there are patterns of vulnerability variation among different bits of application data. Moreover there are patterns when considering the other non-SDC corruption outcomes too. The observed cutoff points of the vulnerability patterns are mainly explained by the different sizes of the tested application data, especially between 8-byte and 16-byte data, and the non-uniform distribution of tests between different data sizes. As our framework tested uniformly on memory load accesses, the number of tests performed per data size reflect the distribution of accesses of these data sizes in the tested applications. In any case, all these point out that there is potential in per-bit eDVF analysis, especially by investigating the eDVF per-bit of

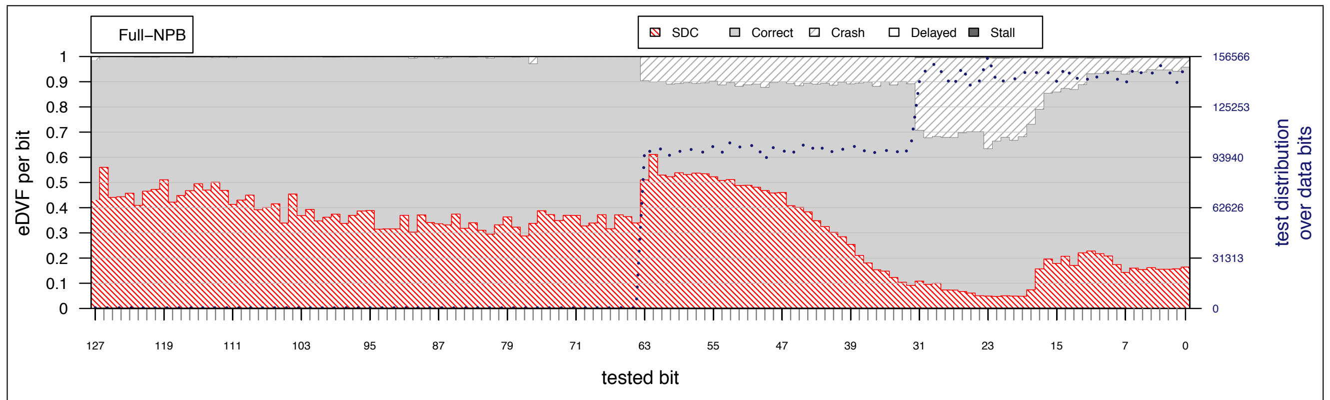

*Figure 3.6: Varying **eDVF per bit** of all tested data for all NPB-serial benchmarks **combined**.*

specific single combinations of data usage types and sizes.

In Fig. 3.7-3.14 the per-bit eDVF analysis is shown for specific combinations of data usage types and sizes. The per-bit eDVF distribution varies among application data usage type categories (as classified in Table 2.1) while, most importantly, it shows consistently common location patterns per each category across most of the tested NPB-serial benchmarks.

For given combinations of usage types and data sizes, the bits that are more likely to result into an SDC tend to be concentrated in continuous bit ranges either at the MSBs (Fig. 3.7-3.8) or at the LSBs (Fig. 3.9-3.14) of application data for most of the tested benchmarks, while the remaining bit ranges have generally near-zero (or a distinctly lower) eDVF per bit making them candidates for reduced fault protection. All these suggest that we can clearly identify bit ranges within particular application data with distinct vulnerability levels to confidently conclude that application data are vulnerable in parts.

More precisely, for the tested floating-point data (FP-8B, Fig. 3.7) the less-vulnerable bit ranges are located at their LSBs across most of the tested benchmarks, while moving towards the MSBs the per-bit eDVF increases steadily. When considering as less-vulnerable bits those with per-bit eDVF less than 0.01, the less-vulnerable bit-range width varies from 20 LSBs for SP up to 32 LSBs for CG, while many of these bits never resulted in an SDC. For the tested FPs, most of the non-SDC observed outcomes were correct executions. The observed bit-range vulnerability variations are explained by the nature of FPs where their LSBs only affect the accuracy of computations, are often discarded by rounding and tend not to affect the outcome. Moving to MSBs it is expected that a corruption there will cause more upset to the data and as

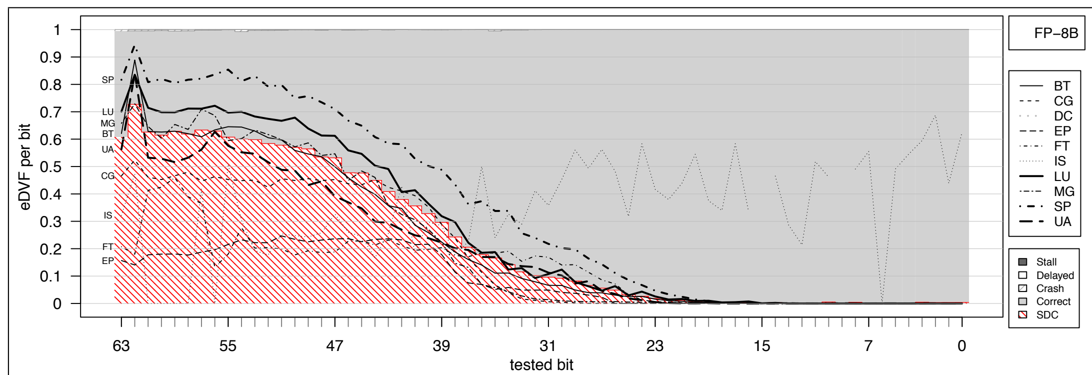

*Figure 3.7: Varying **eDVF per bit** of tested **FP data (8 bytes)** for each NPB-serial benchmark. Background bars show the per-bit outcome breakdown in total over all benchmarks.*

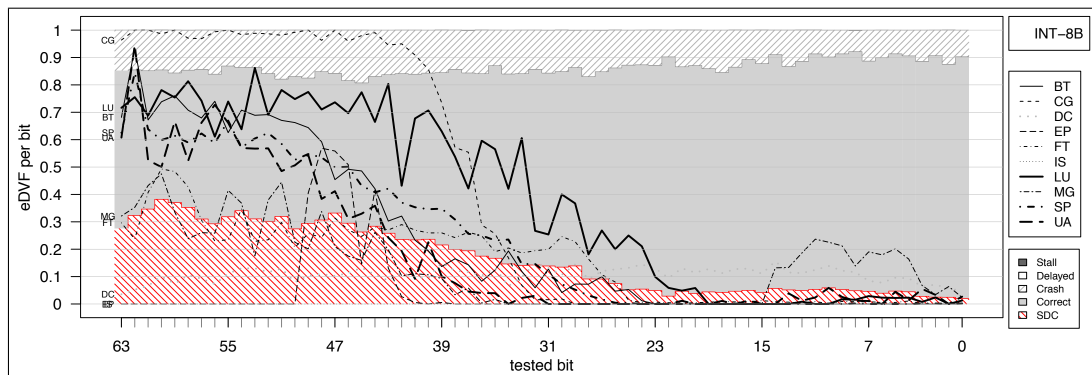

*Figure 3.8: Varying **eDVF per bit** of tested **INT (8 bytes)** for each NPB-serial benchmark. Background bars show the per-bit outcome breakdown in total over all benchmarks.*

such the likelihood of resulting into an SDC increases. This also explains the varying less-vulnerable bit-range width among applications as they have different precision requirements. Moreover, it also explains the observed behavior in IS (Integer Sorting) that is different than the other benchmarks. As FPs in IS are not part of the input/output but are controlling the execution, they are more likely to corrupt the output.

Similar to the FP data, the less-vulnerable bit ranges in the tested INT[2] data are located at their LSBs (INT-8B, Fig. 3.8) but show higher eDVF per bit than their FP counterparts, while the pattern holds for higher eDVFs per bit when moving towards the MSBs. When considering as less-vulnerable bits those with per-bit eDVF less than 0.10, the less-vulnerable bit-range width varies from 24 LSBs for LU up to 43 LSBs for EP (not including DC and IS). This common behavior suggests that data holding values related to the computation, as both FPs and INTs do, tend to corrupt the output

[2]As INT data we denote all non floating-point application data that are not used for addressing memory.

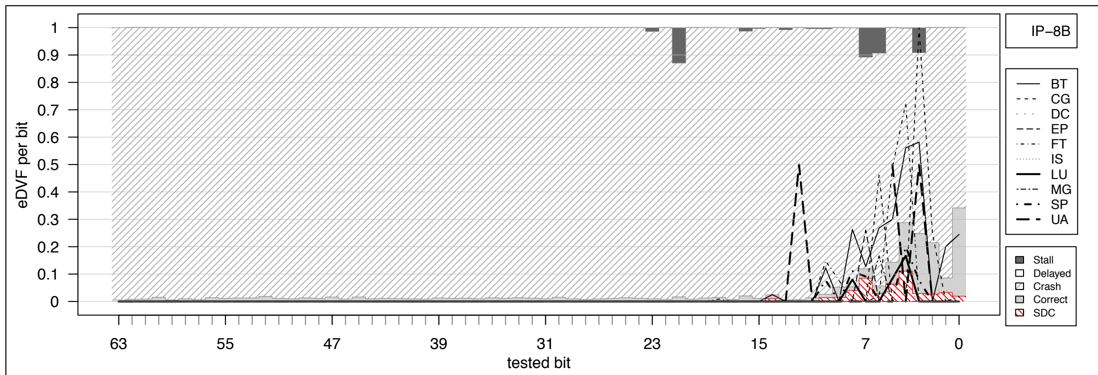

*Figure 3.9: Varying **eDVF per bit** of tested **IP (8 bytes)** for each NPB-serial benchmark. Background bars show the per-bit outcome breakdown in total over all benchmarks.*

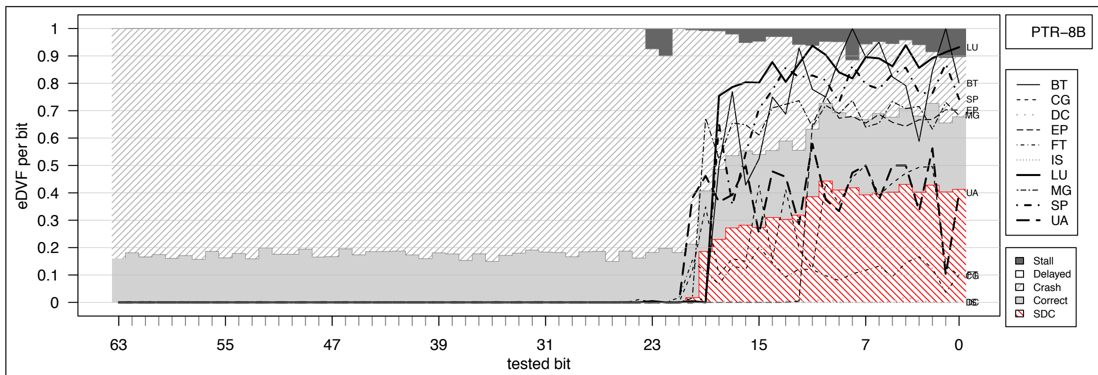

*Figure 3.10: Varying **eDVF per bit** of tested **PTR (8 bytes)** for each NPB-serial benchmark. Background bars show the per-bit outcome breakdown in total over all benchmarks.*

more when they are corrupted at a greater magnitude (i.e., at MSBs). INTs can also be separated into distinct bit ranges with different vulnerability levels. Though, as they are used in many different application specific ways (that we do not yet detect), there is more variation in the width of the less-vulnerable ranges and not a common increasing eDVF pattern in the more-vulnerable bit ranges, as it was the case for the FPs.

Generally, as the data types discussed so far often hold output-related values, corruption at their MSBs tends to result into SDCs and corruption at LSBs into correct executions mostly. Moving on to memory addressing data (Fig. 3.9-3.13), we notice a reversal of the behavior under data corruption where corruption at LSBs tends to result into SDCs and corruption at MSBs into application crashes mostly. This is expected behavior as corruptions in MSBs of memory addressing data will lead to pointers into invalid memory locations and thus cause an application crash. On the contrary, corruptions in LSBs are more likely to lead to pointers into valid memory locations with undesired contents (or incorrect instructions) and thus corrupt the application output (or the

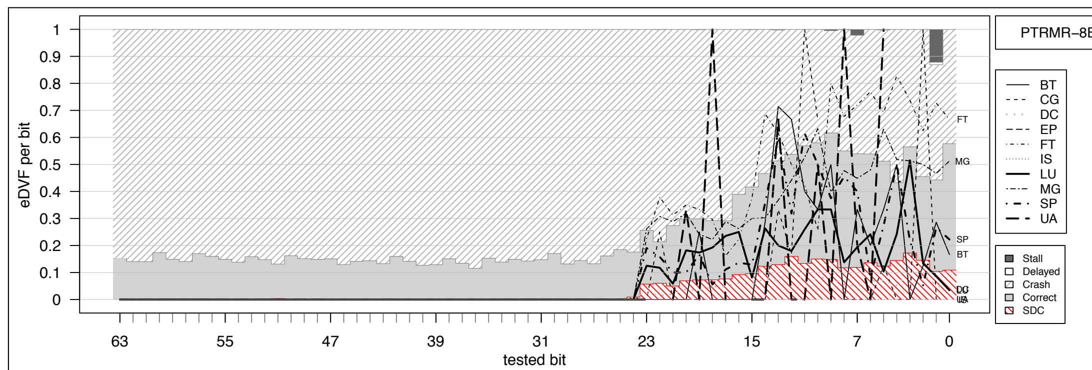

*Figure 3.11: Varying **eDVF per bit** of tested **PTRMR (8 bytes)** for each NPB-serial benchmark. Background bars show the per-bit outcome breakdown in total over all benchmarks.*

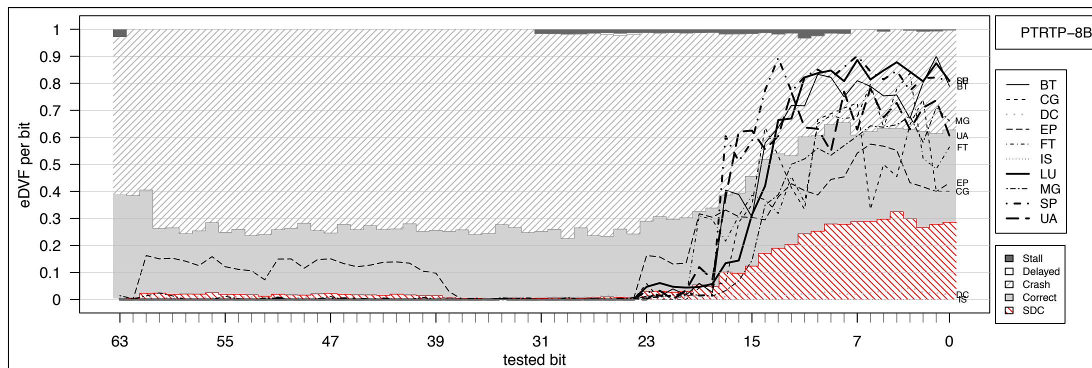

*Figure 3.12: Varying **eDVF per bit** of tested **PTRTP (8 bytes)** for each NPB-serial benchmark. Background bars show the per-bit outcome breakdown in total over all benchmarks.*

instruction flow) but without causing an immediate application crash. This is why the more-vulnerable bit-range width of IP data is narrower than the PTR/PTRMR/PTRTP that are similar. The application program space is narrower compared to the data memory space and, thus, there are less bits in IPs (than in PRT/PTRMR/PTRTP) that if corrupted could still point to a valid location and not cause an application crash but an SDC. Nevertheless, this clear behavior under data corruption for most benchmarks still allows us to identify distinct vulnerability levels within different bit ranges of memory addressing data too.

As shown, it is promising to move into characterizing the effects of data corruption based on the corrupted bit location in specific application data that are classified according to their high-level characteristics (i.e., usage type, size). Especially due to the usage-type-based classification, we were able to get (a) more exploitable insight regarding the vulnerability of usage types that further explains the previous eDVF per usage-type results (Fig. 3.5) and (b) more consistent results among most bench-

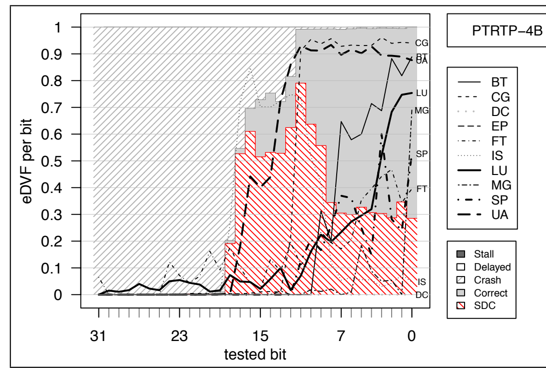

*Figure 3.13: Varying **eDVF per bit** of tested **PTRTP (4 bytes)** for each NPB-serial benchmark. Background bars show the per-bit outcome breakdown in total over all benchmarks.*

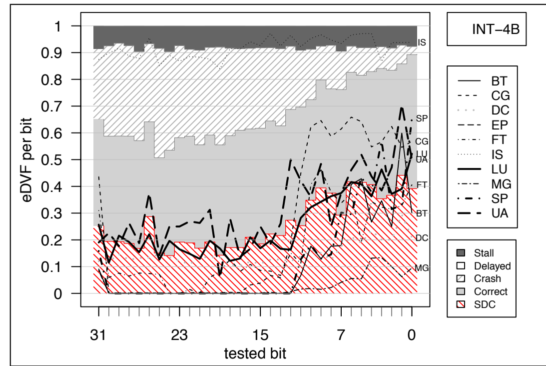

*Figure 3.14: Varying **eDVF per bit** of tested **INT (4 bytes)** for each NPB-serial benchmark. Background bars show the per-bit outcome breakdown in total over all benchmarks.*

marks regarding the location patterns of more-vulnerable application data parts. What changes across benchmarks is the exact width of the less-vulnerable bit range and the vulnerability intensity of the more-vulnerable bit ranges.

Similar analysis could be performed for each of the other identified characteristics of the corrupted data (i.e., location, user) or for the total per-bit eDVFs for all tested benchmarks combined. Such analysis though would not provide the same exploitable insight as the per usage type analysis because the per-bit variation of vulnerabilities depends mostly on how the data-under-consideration are used by the application. If we were to show the bit-level eDVF variation for, e.g., global application data, it would capture the vulnerability characteristics of many different data usage types without helping to make any significant observation. For more detailed insight the eDVF per-bit variation can be also analyzed for combined data characteristics.

As we can now identify clear bit ranges within particular application data with distinct vulnerability levels, the bit-level insight can be used instead of the higher-level

eDVFs to rank application data more accurately. On top of that, the clear partition of application data in more-vulnerable and less-vulnerable parts helps to move further into a bit-level finer granularity of vulnerability-aware protection of application data to further reduce the amount of data under strong protection.

3.4 Memory Space Vulnerability Variation

In the previous section we focused on application data vulnerabilities depending on the data characteristics. In this section, we will extrapolate the testing results to show the vulnerability variations within the tested applications' memory space.

The memory space vulnerability variation is still a data-dependent variation but can offer a broader insight and new exploitation potential compared to the previously discussed vulnerability variations. Memory space vulnerability is a very intuitive area to focus on, especially given the way we performed the fault-injection tests. As a reminder, just before a memory load access, the single bit-flip faults were injected at the loaded memory location and it was straightforward to capture the corrupted data memory address.

Given that this *memory address of the corrupted data* is among the reported corruption characteristics, here we observe the relation between the corruption outcomes and the corrupted data memory address to elaborate on the memory space vulnerability variation. Fig. 3.15-3.16 show the breakdown of the reported corruption outcomes depending on the corrupted memory address for each tested workload.

As each workload does not access the exact same memory space ranges, each workload is shown on a separate figure where only the memory space ranges accessed by it are shown. This includes all memory areas accessed during execution (heap, stack, global/static area). Each point on the horizontal axis is not a single memory location but rather a group of adjacent memory locations plotted against the breakdown of outcomes of the total tests performed within this group of adjacent memory locations. Each memory location may have been tested multiple times depending on the frequency that the application accesses it. The distribution of tests is shown by the blue dotted lines and follows the memory load hotspots during execution. The non-uniform distribution of the tests over the memory space is because the tests were distributed uniformly over the memory load accesses and not the memory addresses.

Out of the figures, it can be seen that the observed outcome breakdown varies over the tested memory space and among the different workloads. More importantly

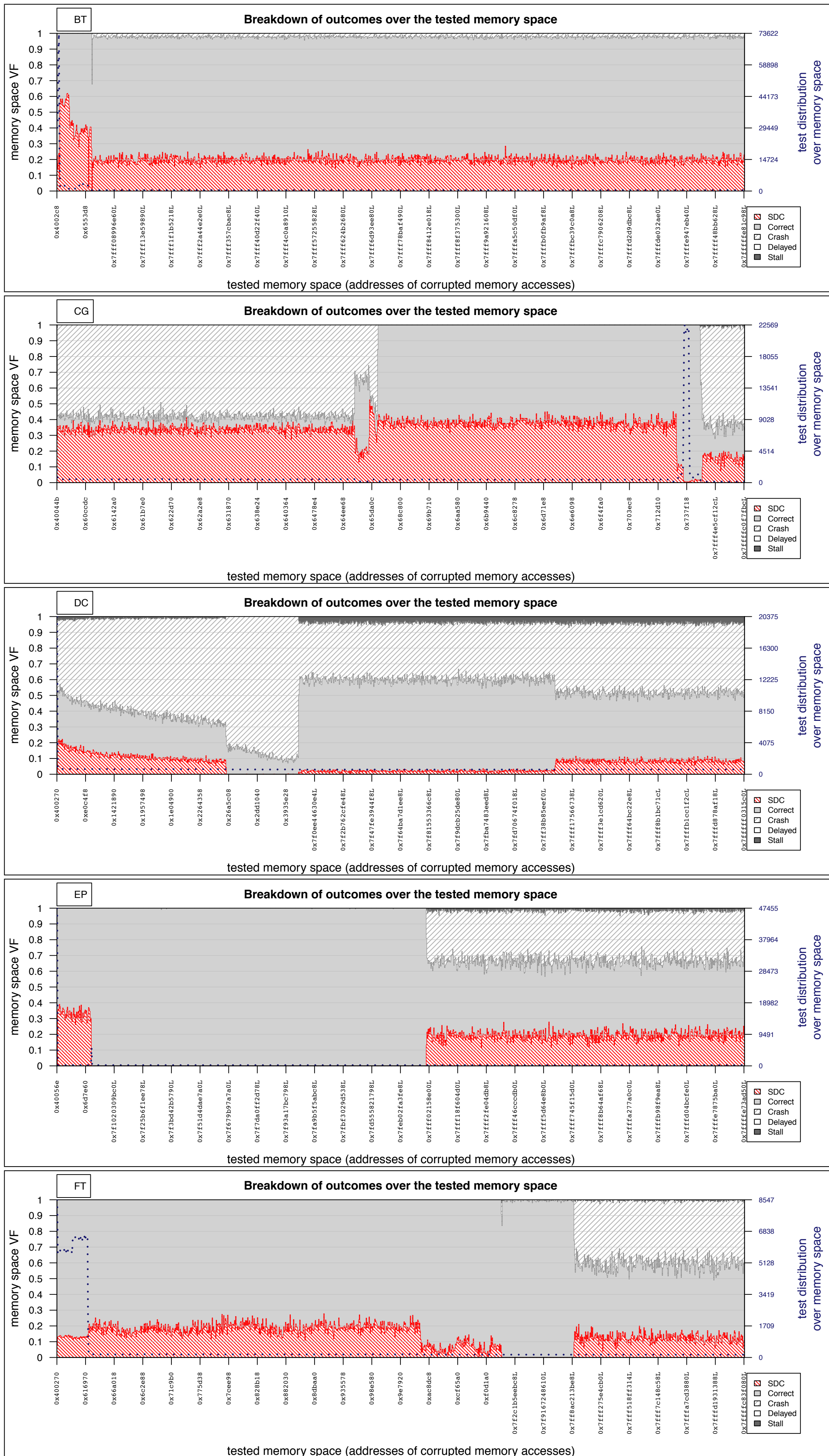

*Figure 3.15: Tested **memory space vulnerability variation** and breakdown of rest outcomes for BT, CG, DC, EP and FT benchmarks.*

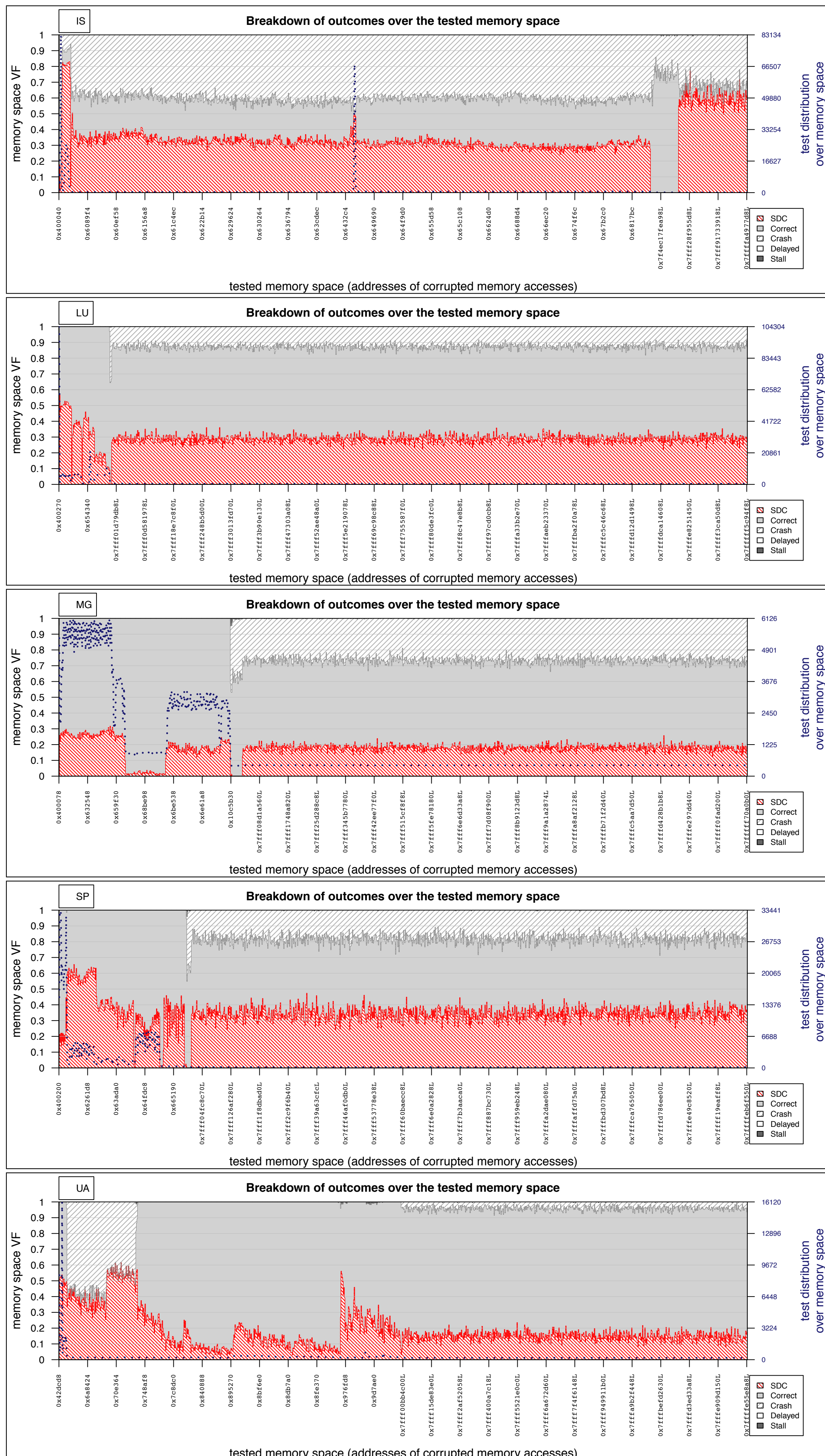

Figure 3.16: Tested **memory space vulnerability variation** and breakdown of rest outcomes for IS, LU, MG, SP and UA benchmarks.

there are distinct memory space ranges with distinctly different corruption outcome percentages. These are mostly attributed to the different memory space areas (heap, stack vs global) and the different types of data mostly occupying that area in each benchmark. Generally there is useful insight in the memory space vulnerability. E.g., there are memory ranges where the only other observed outcome (apart from SDCs) are correct executions only or memory ranges where the non-SDC outcome probability remains constant.

If we focus only on SDCs, the exploitability of the memory space vulnerability still holds. There are still are distinct memory space ranges with distinct vulnerability levels. These can be explained also mainly due to shifting between the global, heap and stack segments and moreover due to the distribution of data types. E.g., the heap of BT is mainly occupied by floating-point data, hence it exhibits a constant vulnerability factor of 0.2 throughout its heap.

To conclude, what is most interesting is that there are again distinct levels of vulnerability over the tested memory space and that provides exploitable insight by various ways. Unfortunately there are no common specific patterns among all benchmarks (at the exact same address ranges) mainly because application data are not distributed in the memory space the same way among applications. Nevertheless the memory space vulnerability variation has potential if it is to be exploited in a per-application basis. E.g., in a hardware reliability approach, the main memory could provide unequal multi-levels of protection where some areas may be protected more if they are more vulnerable to result into an SDC if corrupted. Alternatively, the vulnerability of memory space could be used as a quick way to identify and map vulnerable data to stronger-protected cache banks. Another approach, in a software reliability approach, would be to increase the redundancy and checking of instructions accessing the more-vulnerable memory ranges.

3.5 Register File Vulnerability Variation

So far, the presented vulnerability insight was mainly at application data level and pointed out how to achieve the same protection for less cost at *application data level* only. Given the injected fault model (data corruption at application data directly), it was straightforward to analyze the exact corruption effects in relation to the characteristics of the corrupted data. Although our fault-injection tests were data-level aware, we are not limited to investigating application data vulnerabilities only. Due to the ex-

tra information reported during the fault-injection tests related to the corruption characteristics, we can observe the vulnerability variation of other hardware/software areas too, targeting to obtain more exploitable insight in order to improve their reliability mechanisms too.

In the rest of this chapter, we show what other vulnerability insight we can observe by extrapolating our fault model to model other fault-injection locations by analyzing the reported outcomes to other reported corruption characteristics.

First, in this section, we will extrapolate the reported corruption information to show the vulnerability variations within the register file.

As a reminder of our fault-injection tool operation, once a fault is injected into a memory location just before a memory load access, the corrupted data are eventually affecting an output register through the executing instruction at the moment of fault injection. It is safe to assume that although the fault was injected into a memory location, it is equivalent to being injected into the register file and corrupting that first register. As we capture and report this *first register to load the corrupted data*, we can easily use the existing results to elaborate on the vulnerability of the register file for the tested benchmarks without requiring to remodel/re-execute our tests.

Fig. 3.17 shows the breakdown of corruption outcomes depending on the first register to load the corrupted data for all NPB-serial benchmarks combined. Only the registers that were reported more than 100 times are shown to ensure the confidence of the results. As expected again, the tests are not uniformly distributed over the register file as the registers are not used uniformly to store the accessed data from memory.

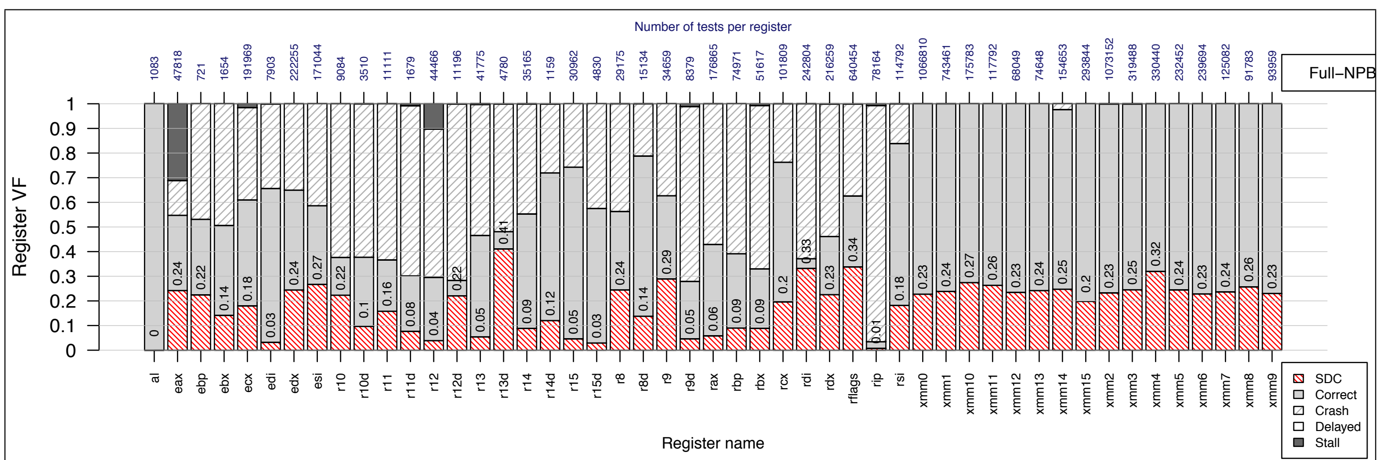

Figure 3.17: Tested ***register vulnerability variation*** *and breakdown of rest outcomes for all NPB-serial benchmarks* ***combined****. Only the registers tested more than 100 times are shown.*

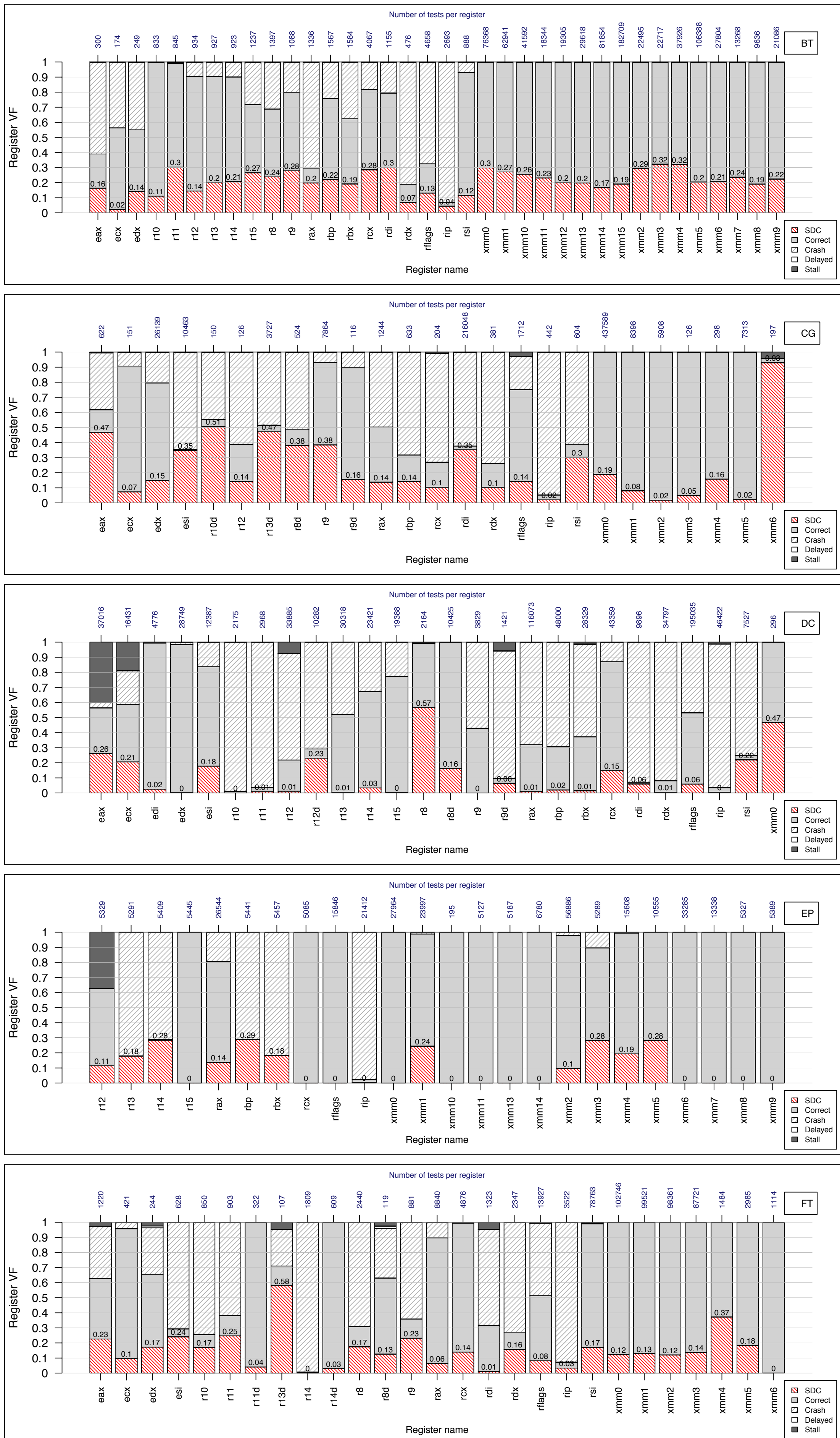

*Figure 3.18: Tested **register vulnerability variation** and breakdown of rest outcomes for BT, CG, DC, EP and FT benchmarks. Only the registers reported more than 100 times are shown.*

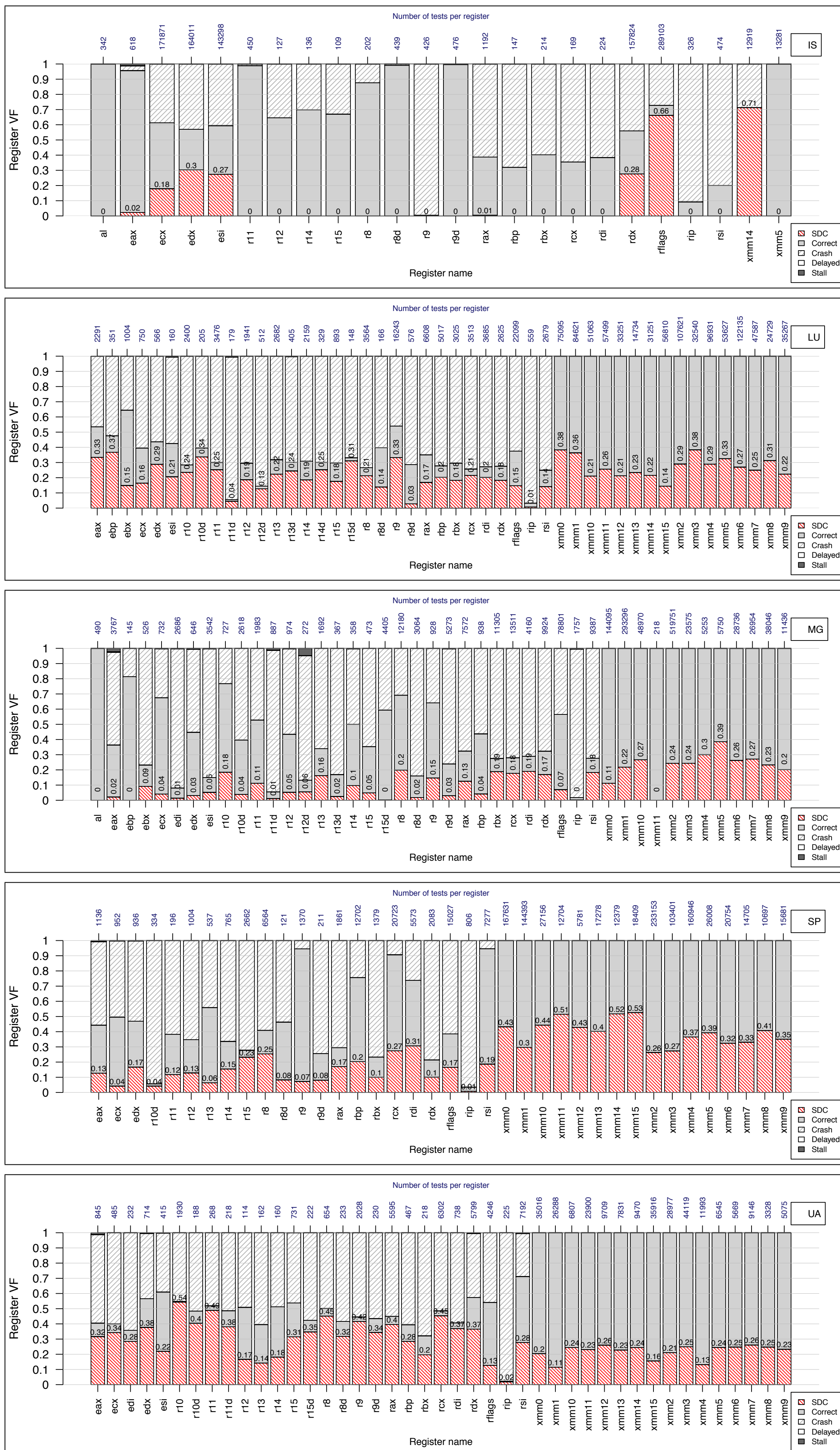

*Figure 3.19: Tested **register vulnerability variation** and breakdown of rest outcomes for IS, LU, MG, SP and UA benchmarks. Only the registers reported more than 100 times are shown.*

The figure also shows the register vulnerability factor per register as a percentage of reported SDC outcomes over the total number of times this register was reported. Although most registers experience SDCs if corrupted, there are registers that experience significantly less SDCs and thus can be deemed as less-vulnerable ones. Interestingly the xmm registers that tend to hold the floating-point data, have a register VF around 0.2, while the rest reported outcomes are almost only correct executions, following the floating-point data observations made in earlier sections.

The observed register vulnerability factors for all benchmarks combined are not safe to be exploited in a benchmark-agnostic fashion because the tests per register type are not distributed equally among the different benchmarks and because a specific register's outcome breakdown varies per application.

Fig. 3.18-3.19 depict the breakdown of corruption outcomes depending on the first register to load the corrupted data per each tested benchmark. The only common behavior is regarding the xmm registers (except for IS) that tend to have similar vulnerability factor per benchmark, while the rest reported corruption outcomes tend to be almost only correct executions. This, once more, is explained by the high occupancy of xmm registers by floating-point data.

Despite the non benchmark-wide observations, a per-application profiling can still point to some exploitable insight. The registers can be ranked per-application according to their vulnerability in order to assign their protection strength accordingly. As the register file is relatively small there would be not much benefits in adapting an existing register file hardware protection mechanism.

On the contrary, there is more potential for savings in software-level reliability mechanisms that could become aware of which registers tend to hold data that tend to be more vulnerable. In such a scenario, a software-level reliability mechanism could take care to replicate the data stored in such registers and also drive the more vulnerable data to the registers usually occupied by more vulnerable data to minimize the vulnerable area of the register file. The reverse intuitive approach of targeting to use more the less-vulnerable registers would not work as intended because the registers' vulnerability reflects the vulnerability of the data that they hold.

3.6 Instruction-Level Vulnerability Variation

In a similar fashion as before, in this section we will extrapolate the original injected fault model to model instruction-level faults to observe vulnerability variations in re-

lation to instruction-level characteristics. Assisted by the reported corruption information relating to the instruction at the moment of fault injection and the instruction using first the corrupted data, we will discuss on instruction-level and program space vulnerability variations.

Although we injected faults at memory locations just before a memory load access, it is possible to infer instruction-level vulnerabilities. Due to the reported information of our fault-injection tool, we can collect information such as (a) the instruction opcode at the moment of the fault injection, (b) the instruction opcode of the first instruction to use the first affected register, (c) the instruction pointer at the moment of the fault injection and (d) the instruction counter at the moment of the fault injection.

All these were straightforward to be collected by our fault-injection tool and make up some interesting characteristics to be investigated hereafter in this section. In particular, we will (a) model the vulnerability of instructions if they are targeted by a fault, (b) model the vulnerability of instructions if they use a corrupted operand, (c) model the vulnerability variations within the program space and (d) observe possible program vulnerability phases.

3.6.1 Per Instruction Type at Fault Injection

During our fault-injection tests, the faults were injected just before instructions that caused a memory load operation. As we capture the opcode of that instruction, we can elaborate on the vulnerability variation at instruction level. The vulnerabilities that are

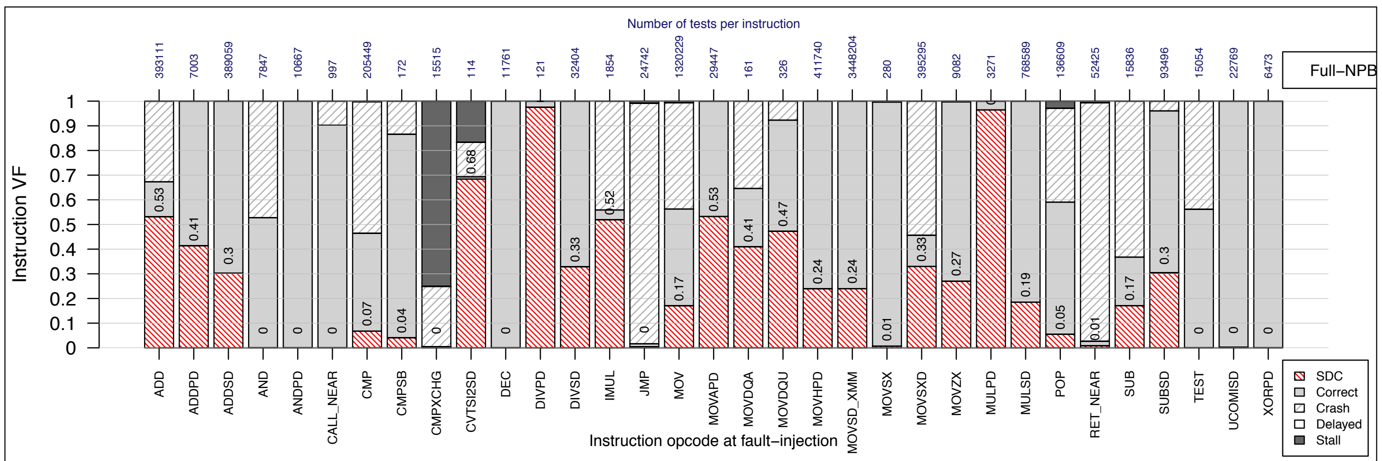

*Figure 3.20: Tested **instruction vulnerability variation at fault injection** and breakdown of rest outcomes for all NPB-serial benchmarks **combined**. Only opcodes reported more than 100 times are shown.*

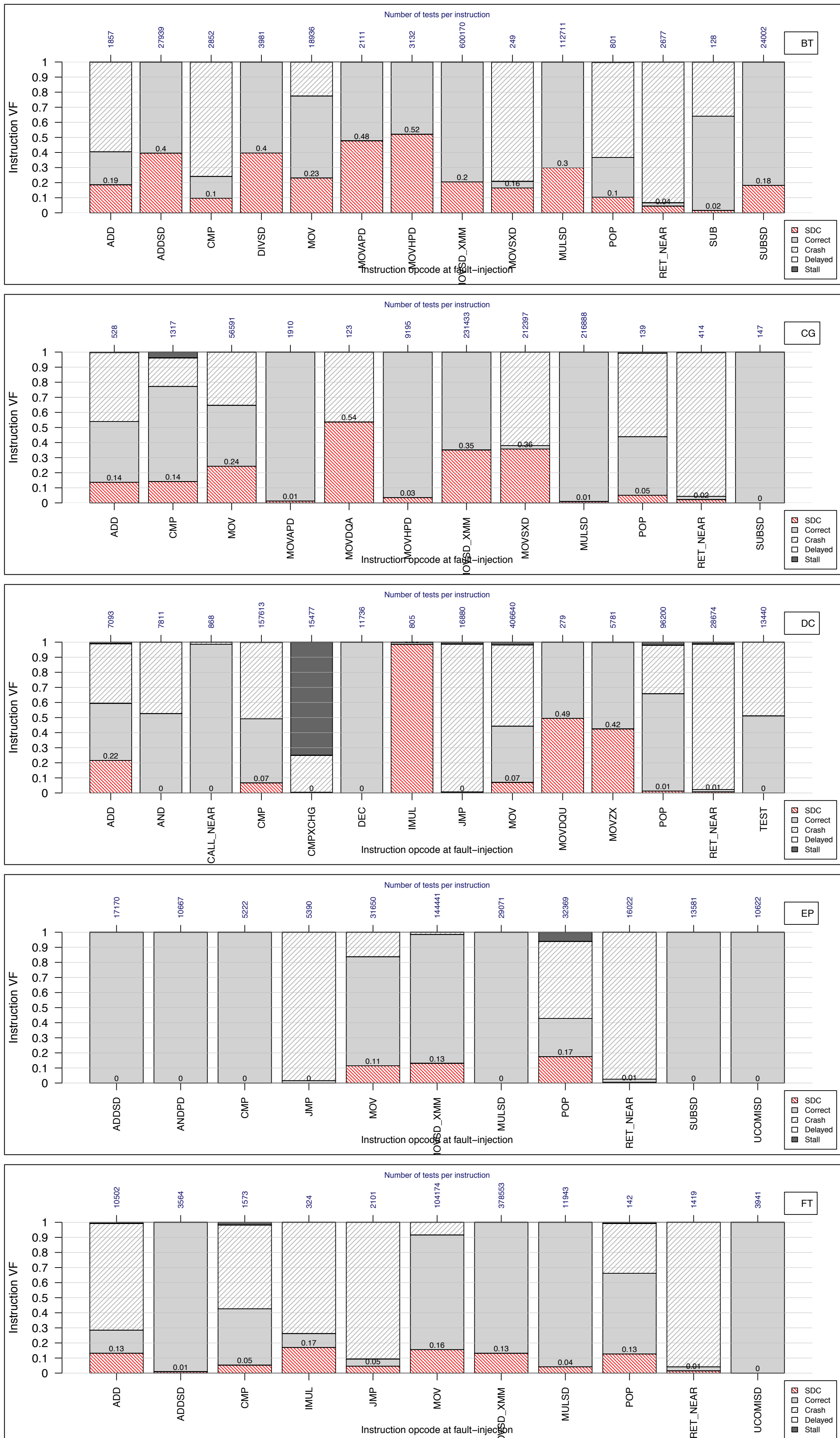

*Figure 3.21: Tested **instruction vulnerability variation at fault injection** and breakdown of rest outcomes for BT, CG, DC, EP and FT benchmarks. Only opcodes reported more than 100 times are shown.*

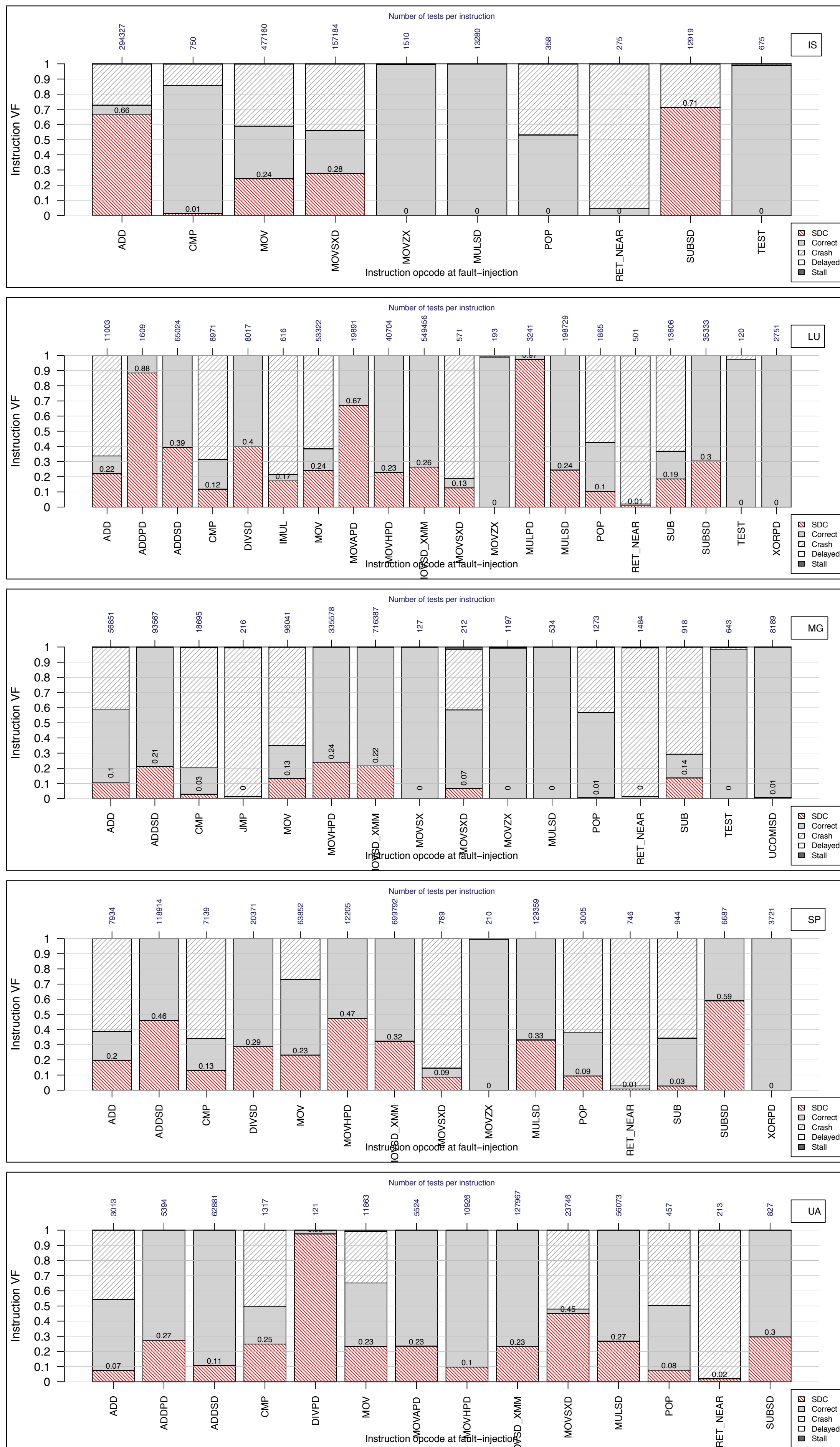

*Figure 3.22: Tested **instruction vulnerability variation at fault injection** and breakdown of rest outcomes for IS, LU, MG, SP and UA benchmarks. Only opcodes reported more than 100 times are shown.*

modeled using that information are only for *instructions that load corrupted data from memory*. Such instructions may either use the corrupted data immediately (e.g., adding a register value to a value from memory) or move the corrupted data from memory.

Fig. 3.20 shows the breakdown of corruption outcomes depending on the instruction type at the moment of fault injection for all NPB-serial benchmarks combined. Only the instruction opcodes that were reported more than 100 times are shown to ensure the confidence of the results, while the distribution of the tests over the reported opcodes is also shown. The figure also shows the instruction vulnerability factor per opcode as a percentage of reported SDC outcomes over the total number of times that this opcode was reported.

Although this considers only instructions that load corrupted data and, thus, limits the instruction types under consideration, it still provides an indication of instruction-level vulnerabilities. Ranking such instructions based on their probability to result into an SDC, if they load corrupted data, can be used as a guideline on which instructions to protect stronger. This also holds if we consider that this can model the corruption happening inside the functional unit implementing the given instruction; e.g., the outcome breakdown of an ADD where one of the operands has a corrupted bit is the same with the outcome breakdown where the operands are correct but a single output bit is wrongly calculated.

The tests are not uniformly distributed over the instruction types and, as before, the presented instruction-level vulnerability for all benchmarks combined is not safe to be exploited in an benchmark-agnostic fashion. An exception holds for those clear results that are explained due to the instruction type itself, e.g., the non-vulnerable instructions AND and TEST that have zero vulnerability factors across all benchmarks.

More usable insight can be gained in a per-application profiling. Fig. 3.21-3.22 show the same breakdown of execution outcomes per each tested benchmark. It still holds that instruction types may be ranked depending on their vulnerabilities in a per-application basis.

If we focus only on the reported SDC percentages, it is interesting that there are clear vulnerability variations among different opcodes, while some of them never reported an SDC despite the large number of times reported. This points out that it is promising to rank opcodes based on their vulnerability so that we employ more measures to ensure that a memory-loaded value has been correct for those instruction opcodes with higher vulnerability to result into an SDC. Similarly, if we assume that the corruption occurs within the functional/arithmetic units that implement the instruction

(only for instructions that do indeed use the corrupted data), then this can point out which units to protect stronger than others. Equally interesting exploitation potential lies with these instructions that never reported an SDC.

3.6.2 Per Instruction Type at First Consumption of Corrupted Operand

During our fault-injection tests, the faults were injected just before instructions that caused a memory load operation. Then these instructions could either use the loaded corrupted data immediately or just move them from the memory to a register. In both cases, there is always an output register that is affected by these instructions.

Here we target the instruction-level vulnerability variations of the *first consumer of the corrupted data*. In other words, we observe the instruction-related vulnerability to result into an SDC, if the instruction is the first to use the first affected register as an input register *after* the instruction at fault injection.

To achieve this we need the opcode information of this first consumer of the corrupted register that is provided by our tool. The process of obtaining this is different than the other instruction-related captured information. To capture the instruction opcode, instruction pointer and instruction counter at the moment of fault injection was straightforward.

On the contrary, to capture the intended instruction opcode required tracking the first register, that the corrupted data affect first, until its first use as an input register

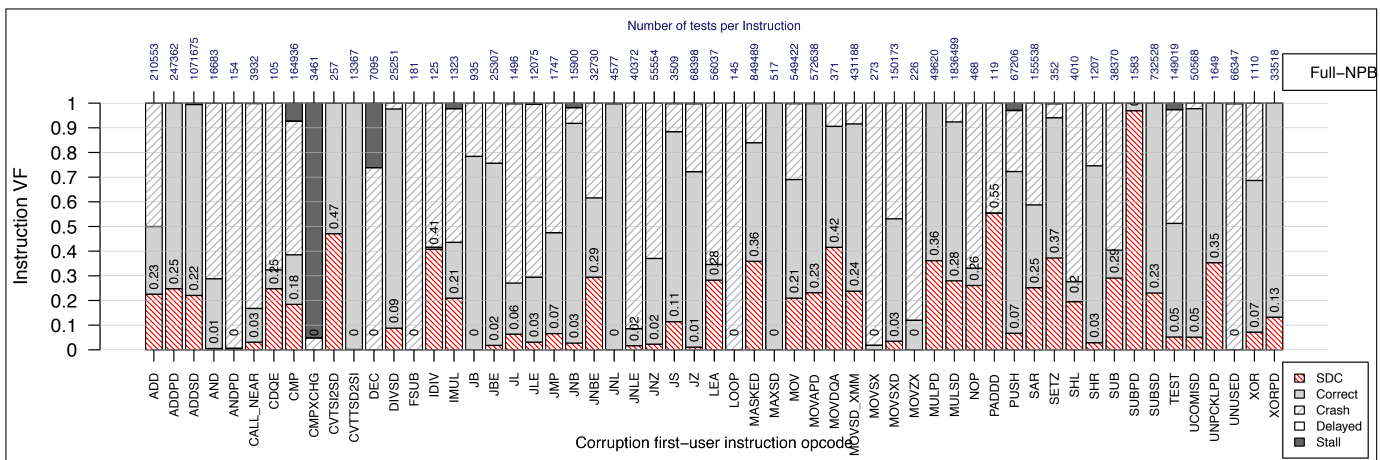

*Figure 3.23: Tested **instruction vulnerability variation (at first consumption of corrupted operand)** and breakdown of rest outcomes for all NPB-serial benchmarks **combined**. Only opcodes reported more than 100 times are shown.*

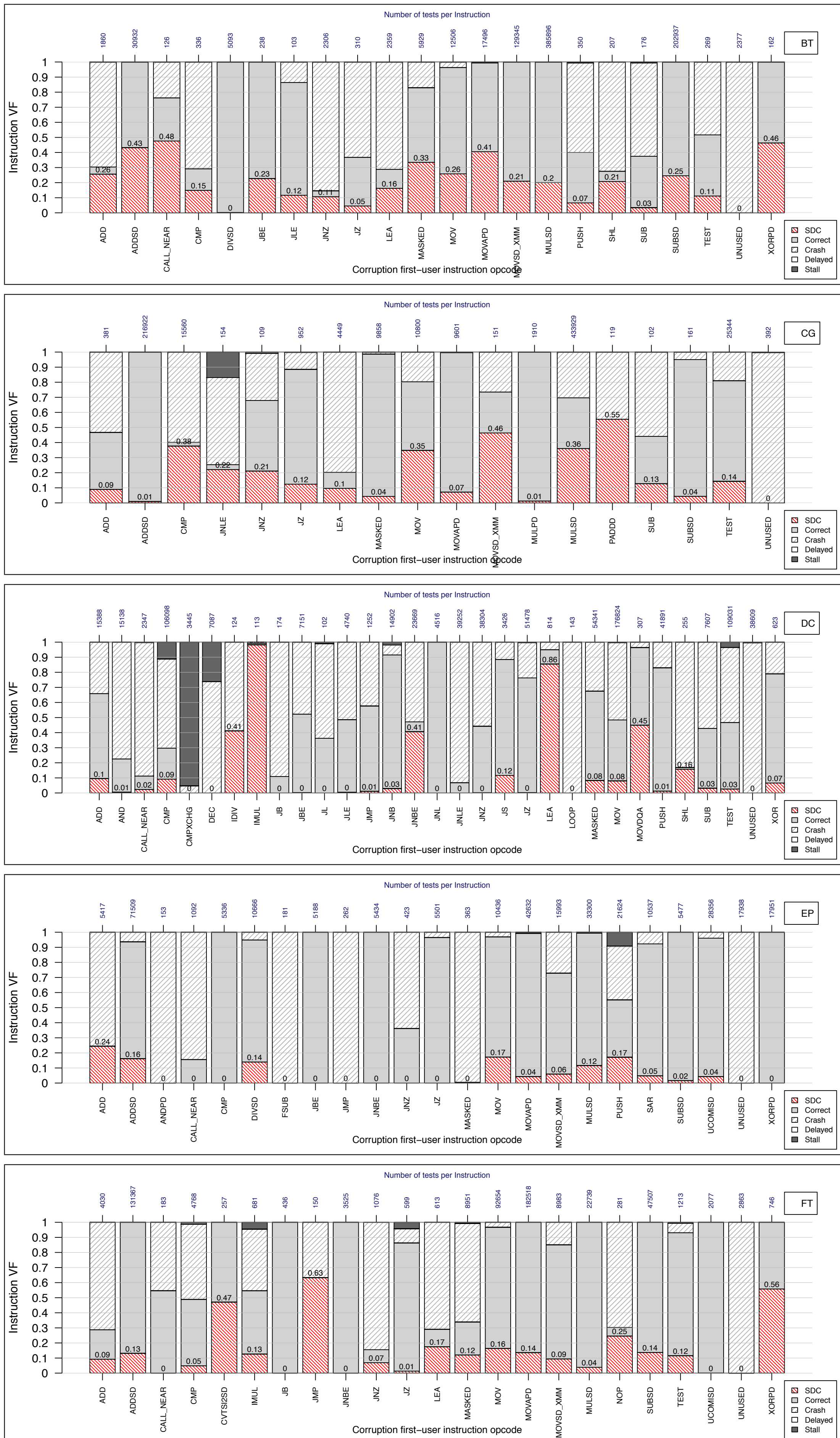

Figure 3.24: Tested ***instruction vulnerability variation (at first consumption of corrupted operand)*** *and breakdown of rest outcomes for BT, CG, DC, EP and FT benchmarks. Only opcodes reported more than 100 times are shown.*

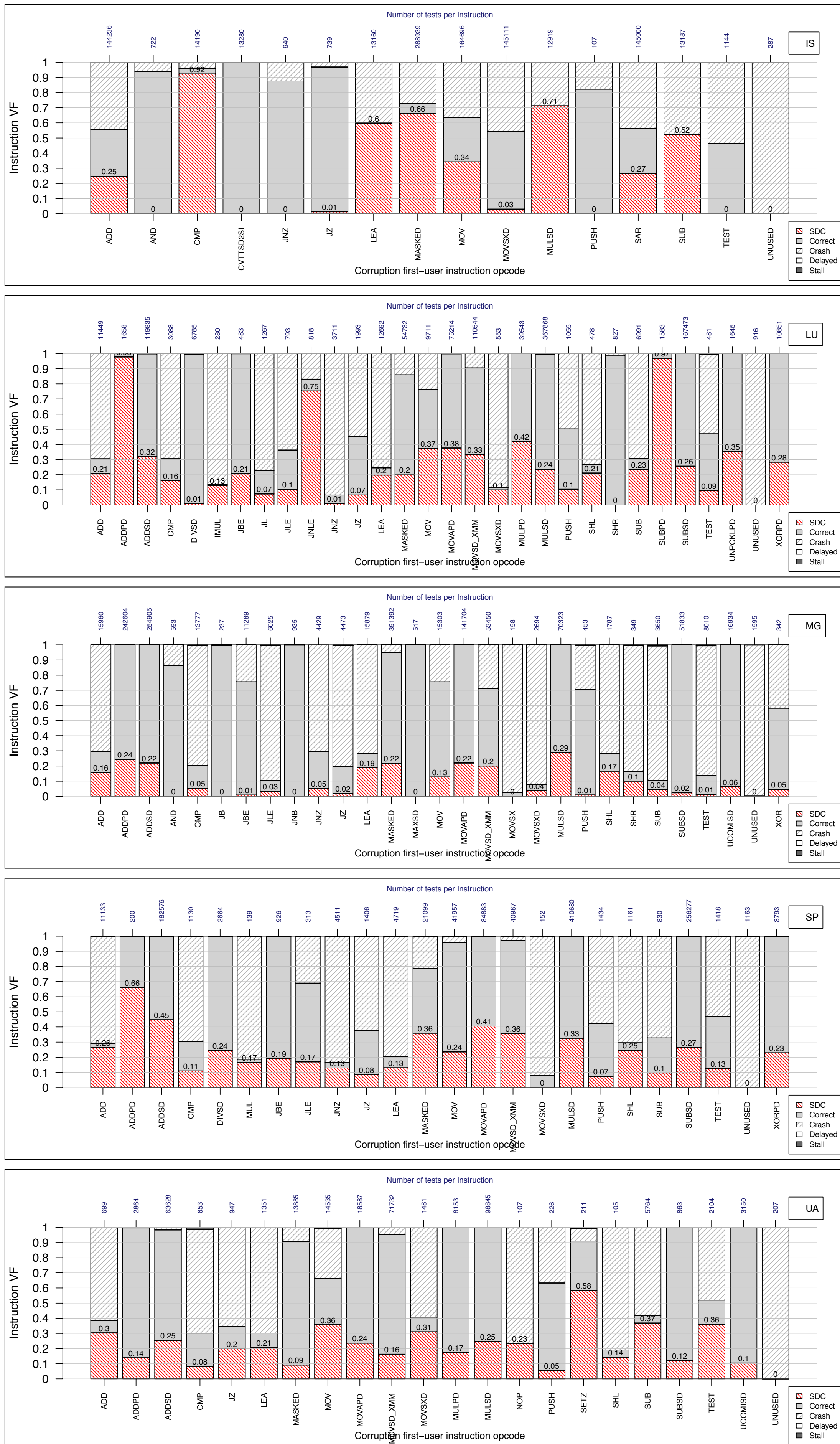

*Figure 3.25: Tested **instruction vulnerability variation (at first consumption of corrupted operand)** and breakdown of rest outcomes for IS, LU, MG, SP and UA benchmarks. Only opcodes reported more than 100 times are shown.*

by an instruction. This tracking of the first affected register by the corruption is for a window of 1000 instructions. If by the end of this instruction window the register hasn't been used by a later instruction as an input register or if the application crashes first, then this case will be reported as an 'unused' case.

As there is the possibility of this register being overwritten before its first use as an input register, the tool makes sure to monitor and report such cases as 'masked'. This doesn't mean that the initial memory corruption got masked but that the register where the corrupted data first got loaded got masked. The corrupted data will still reside in the memory, unless overwritten in the meantime.

There is also a special common case where, at fault injection, when the corrupted data are loaded from the memory, they do not get stored to a register but the outcome of their processing is stored to one of the bits of the flags register. In that case the flags register is assumed as the first register affected by the corrupted data. As such it is very likely to be overwritten by a later instruction and being reported as another 'masked' instance.

As all these help gather information about the first instruction to use the first affected register by the initial corruption, we can show the variation of corruption outcomes depending on the opcode of the first instruction after the fault injection to use the first affected register. Fig. 3.23 shows this outcome breakdown for all NPB-serial benchmarks combined, along with the observed instruction vulnerability factors. Again, only the instruction opcodes that were reported more than 100 times are shown, along with the distribution of tests over the reported opcodes.

It may be observed that different instruction opcodes have different vulnerability to result into an SDC if they are the first to consume the affected register. This points out that there is exploitation potential. Although the corruption does not happen within the instruction types under consideration, this type of instruction vulnerability can point out where extra protection needs to be added to ensure that the vulnerable instruction gets correct data at their input registers.

If the same vulnerabilities are observed per tested benchmark (Fig. 3.24-3.25), there are more clear areas of where to focus. There are more instruction types that are more invulnerable to faults in their input operands and there is still a clear variation to the rest instructions, although this variation does not hold across benchmarks. For the instance of the reported 'unused' cases, it can be seen that all such cases result only into crashes and not any SDCs, meaning that the crash happened before the register was used again by any instruction.

Despite the non benchmark-wide observations, using the vulnerability of instruction types when one of their operands is corrupted can be exploited in various ways. E.g., there is potential in a software-level reliability mechanism that focuses on a stronger check of the correctness of input operands only for the more-vulnerable instruction types.

3.6.3 Program Space Vulnerability

Given that our fault-injection tool captures and reports the *instruction pointer of the instruction executing at the moment of the fault injection*, we can use this information to relate the corruption outcomes to the instruction pointer value at fault injection in order to observe the tested benchmarks' *program space vulnerability variations*.

Fig. 3.26-3.27 show the breakdown of the reported corruption outcomes over the program space for each tested workload. As each workload does not reside in the exact same program space ranges, each one is shown on a separate figure where only the instruction locations that triggered a fault injection are shown. This includes both the workload's space and the used libraries' space. Each point on the horizontal axis is a group of adjacent memory locations storing instructions plotted against the breakdown of outcomes of the total tests performed within this group of instructions. The distribution of tests over the program space is shown by the blue dotted lines. As it follows the program execution hotspots (of instructions that cause memory load accesses) it resulted into a non-uniform test distribution once again.

The target is to notice distinct ranges within the program space with distinct vulnerability levels and distinct breakdown of the rest corruption outcomes. Generally, it can be seen that the bottom parts of the program space, where the library/system binaries are loaded into, have near-zero occurrences of SDCs (as shown too in Fig. 3.4) and tend to result into either a correct execution or an application crash when a fault is injected during their execution.

On the contrary, most SDCs occur at the workload's space but there is not a common pattern to be observed among workloads. Still though there is room for exploitation for single workloads where there appear to be continuous ranges with higher vulnerability than other ranges. This implies that there are application code parts that can be collectively marked as more vulnerable instead of marking individual instructions as such.

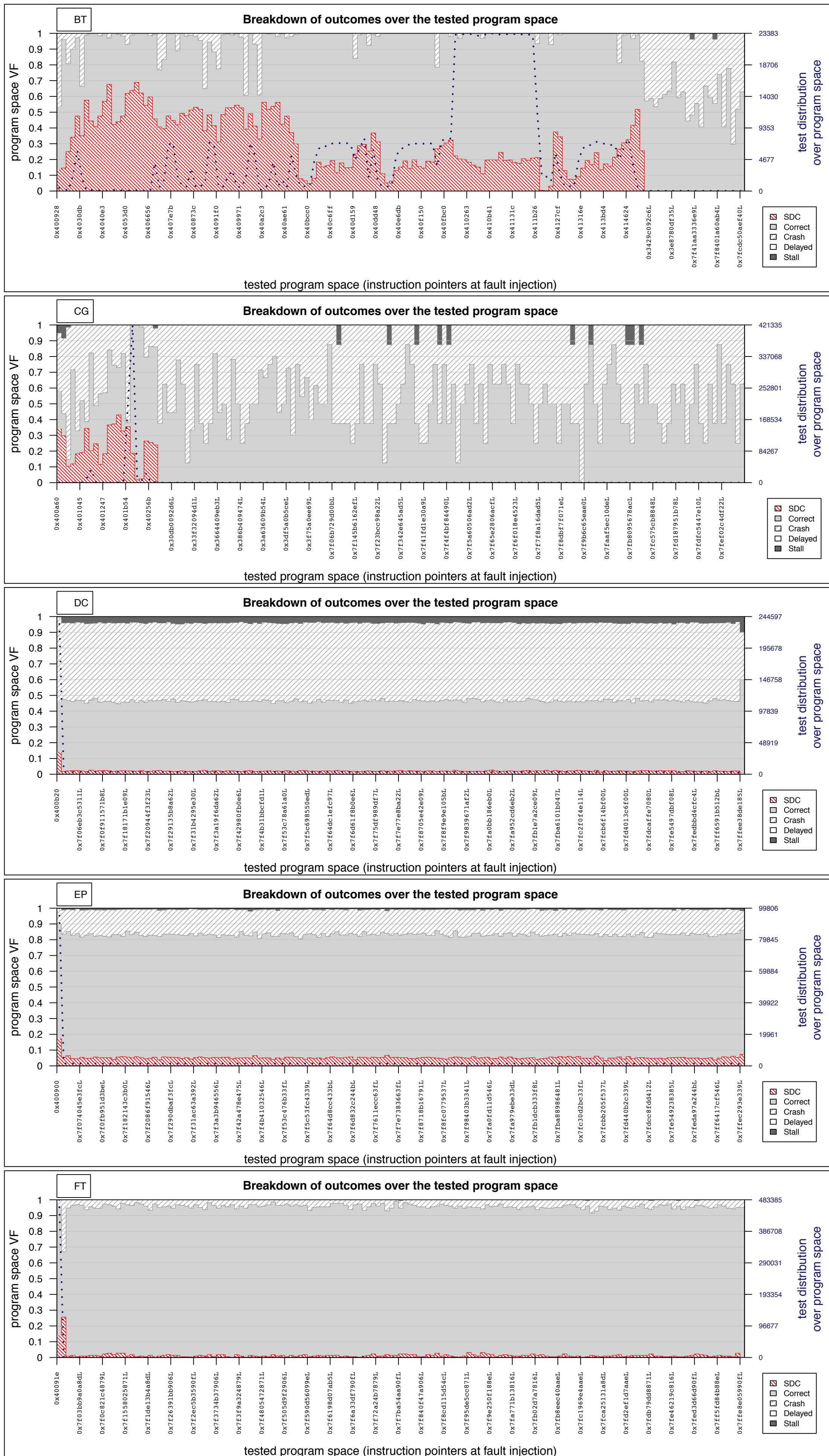

*Figure 3.26: Tested **program space vulnerability variation** and breakdown of rest outcomes for BT, CG, DC, EP and FT benchmarks.*

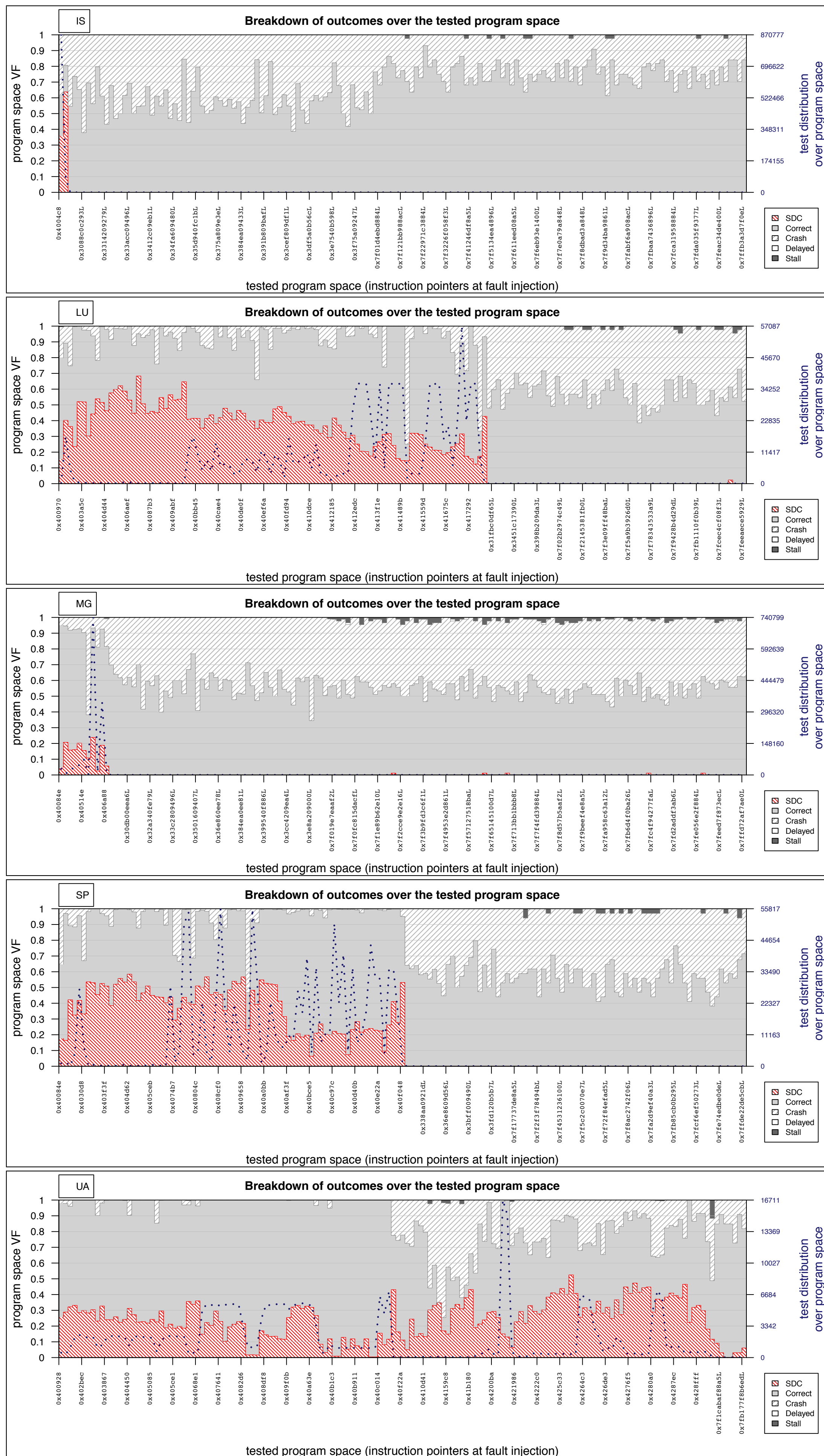

*Figure 3.27: Tested **program space vulnerability variation** and breakdown of rest outcomes for IS, LU, MG, SP and UA benchmarks.*

3.6.4 Program Vulnerability Phases

As the just presented program space vulnerability does not account for the program phases, here we look into possible program phase related vulnerability patterns. As our fault-injection tool reports back the *instruction counter at the moment of fault injection*, it is straightforward to observe the outcome breakdown of corruptions over the lifetime of an executing workload to identify possible *program vulnerability phases*.

Fig. 3.28-3.29 show the breakdown of the reported corruption outcomes against the time the fault injection occurred (as measured by the instruction counter). As before, each workload is shown separately. Unlike before, the test distribution is uniform because we injected the faults in a way to distribute them uniformly over the memory accesses that occur uniformly over time.

Out of the figures it can be seen that in most benchmarks there are clear repeating patterns over time per benchmark for all reported corruption outcome occurrences. This indicates that an application may have distinct program vulnerability phases that mostly reflect the program execution phases. For most of the tested applications they appear to be such vulnerability phases per application at the beginning of the execution, at the end of the execution and one repeating in-between.

It is interesting that these outcome patterns are present for all types of reported outcomes. What usually tends to vary, apart from the pattern's strength, is the length of the repeating pattern over time; some benchmarks have phases longer than other. Moreover, it is highly useful that there are application intervals that don't exhibit all corruption outcomes, e.g., in DC and MG there are long time intervals where almost zero SDC percentages were observed.

Moving on to considering only the SDC occurrences, there are still vulnerability patterns to indicate program vulnerability phases. As it was expected, there are no common vulnerability phases across all benchmarks, indicating that there is more potential in a per-application basis.

Identifying program vulnerability phases has great exploitation potential. As it points out when, during the execution, an application becomes more vulnerable, it can drive a time-aware reliability mechanism that strengthens during specific time intervals. Moreover, especially for repeating program vulnerability phases, it can be used to prepare protection strategies for a particular vulnerability phase pattern and deploy the strategy repeatedly to deal with the specific vulnerability phase footprint. Similarly, it can be used to break an application into vulnerability phases where each one has a

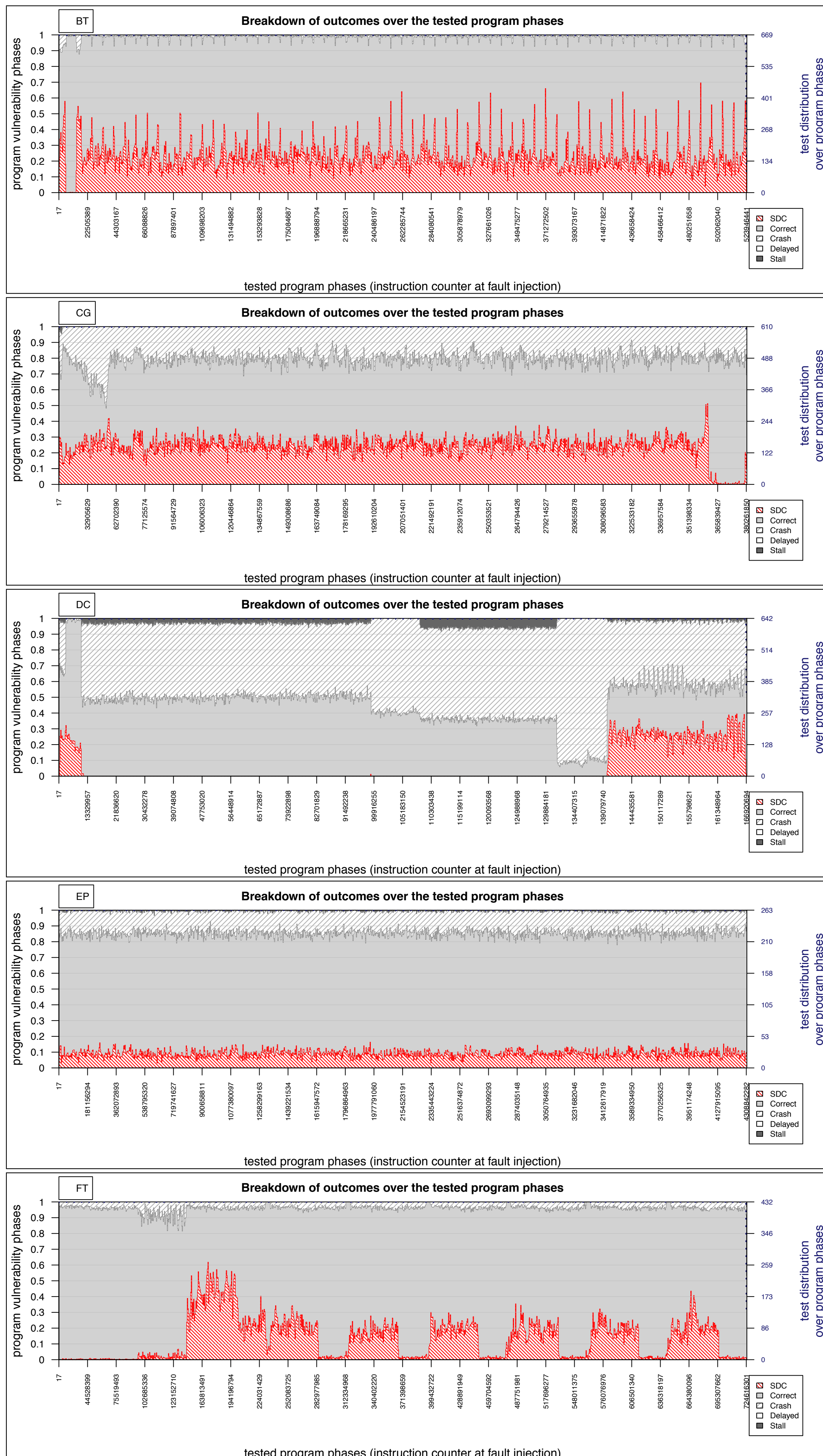

*Figure 3.28: Observed **program vulnerability phases** and breakdown of rest outcomes per program phase for BT, CG, DC, EP and FT benchmarks.*

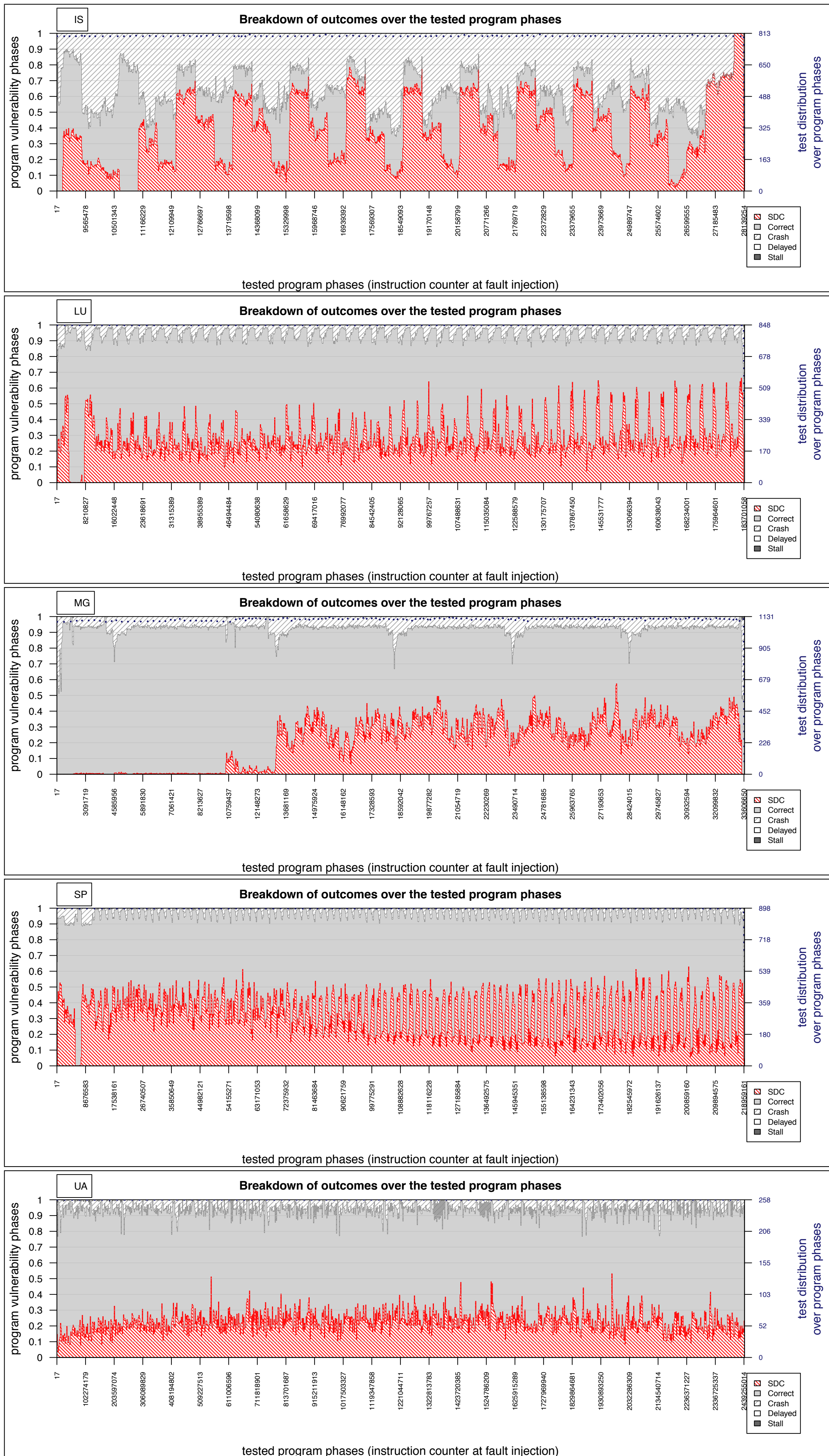

*Figure 3.29: Observed **program vulnerability phases** and breakdown of rest outcomes per program phase for IS, LU, MG, SP and UA benchmarks.*

different protection strategy applied to it depending on the vulnerability phase pattern. Finally, identifying program vulnerability phases could be used to reduce the cost of evaluating reliability mechanisms. This holds true especially for the identified repeating phases, where one can avoid repeatedly testing a repeating vulnerability phase, by testing only one instance of it.

3.7 Summary

In this chapter, we observed how application behavior varies under data corruption depending on the characteristics of the corruption and the executing workload.

For that purpose, using our SWIFI framework, we setup and performed an excessive testing of the NPB-serial benchmark suite comprising of 7.8 million fault-injection tests in total (Section 3.1). First we presented the top-level workload-related vulnerability variations in NPB-serial (Section 3.2) that indicated that applications have different inherent vulnerability characteristics.

The amount of performed tests and the reported information enabled us to further our investigation at data level to observe the application data related vulnerability variations (Section 3.3). We showed how the application behavior in NPB-serial benchmarks varied depending on the high-level characteristics of the corrupted application data; in particular, in relation to the (a) size, (b) usage type, (c) user and (d) memory space location of the corrupted data.

As generally there were no application-independent vulnerability observations to be made, we moved on to a finer-granularity characterization where we showed how the application data vulnerability in NPB-serial benchmarks varied depending on the exact bit location of the injected corruption (Section 3.3.1). This allowed to safely identify distinct bit ranges within application data types where the probability to result into an SDC, if corrupted, is very low.

Then we moved beyond to elaborate on areas of vulnerability other than application data related. Given the reported information by our fault-injection tool (relating to the corruption characteristics and the corruption effects) and our original fault model (single bit flips at memory locations just before memory load accesses), we extrapolated the original fault model to model as many other locations of corruption as possible without the need to remodel and repeat the experiments. This enabled to observe the vulnerability variations of NPB-serial benchmarks within the memory space (Section 3.4), the register file (Section 3.5) and among instruction-level characteris-

tics (Section 3.6), including instruction type vulnerability variations, program space vulnerability variations and program vulnerability phase detection.

Out of all the observed vulnerability variations the most interesting ones related to the per-bit variations within application data. These pointed out that it was useful to move in a *bit-level* and *data-aware* investigation of vulnerabilities as what drives the vulnerability of data is mostly the way they are used by the application.

We showed that, when considering combinations of data usage types and data sizes, there are distinct bit ranges within application data with distinct vulnerability levels. This led to conclude that application data are vulnerable in parts. The data types holding output-related values have less-vulnerable continuous bit ranges at their MSBs, while memory addressing data at their LSBs. E.g., each of the 32 LSBs of FPs in CG has less than 1% probability to cause an SDC if corrupted. As the observed less-vulnerable bit ranges were wide and experienced near-zero SDCs, they showed promise for a high exploitation potential, as we will establish in the next chapter.

Moreover, this observed vulnerability of data in parts showed high exploitability potential, among other reasons, also due to holding true for most of the tested benchmarks. Generally very few application-independent observations could be made relating to the other investigated vulnerability variations. Despite that, the characterization results still showed exploitation potential in various locations (as it was mentioned throughout this chapter), even if they are to be used in a per-application basis assuming a previously vulnerability-characterized application. Nevertheless, all results pointed out that application behavior can be characterized based on the characteristics of the corruption and the executing application.

In the next chapter we will investigate on the potential benefits of exploiting the gained characterization insight in order to reduce the overhead of reliability mechanisms. In particular, out of all the observed vulnerability insight, we are going to focus on exploiting the (per-bit) application data related vulnerability variations.

Chapter 4

Exploiting Application Behavior Characterization

In this chapter we build upon the gained characterization insight from Chapter 3 in order to optimally exploit the application behavior under data corruption.

In particular, we will show the *potential benefits* of exploiting *the per-bit data-level vulnerability variations in a vulnerability-driven manner targeting the reliability-overheads reduction in a reduced-cost unequally-protected data cache that offers two levels of fault-protection strength while running the NPB-serial benchmarks.*

Among the insight of the characterization study, data-level vulnerability factors were estimated for a given workload, down to identifying distinct bit ranges with different vulnerability levels within specific types of application data. This can be intuitively exploited by shifting to vulnerability-aware unequal-protection architectures where the protection strength is driven by the application data vulnerability. Our intension is to introduce the unequal protection within a given structure or same-level structures and not among different structures.

We argue that such a shift is beneficial as it reduces the fault-protected surface for a given reliability QoS level and, as such, reduce the reliability costs by avoiding excessive strong protection of the less-vulnerable application data. Considering that our characterization study goes down to bit-level vulnerability estimation, the performance/cost benefits can be maximized for an even smoother trading off against reliability. This is contrary to traditional reliability approaches that either unnecessarily protect all parts equally and exhaustively [16, 19, 52, 11] or offer unequal protection agnostic to the actual distribution of likelihood of the application's data to corrupt the

execution output [55, 22, 23].

As mentioned throughout Chapter 3, the various characterization insight shows numerous ways of exploitation potential depending on the observed area of vulnerability variations and the design targets. Therefore, in Section 4.1 first we justify why out of all the alternative options we decided to exploit that particular characterization insight in that particular way and then we discuss on the desired characteristics for identifying areas of high exploitability potential (Subsection 4.1.1) to maximize the potential benefits.

Then in Section 4.2 we describe how we profiled the contents of a data cache running the NPB-serial benchmarks in order to obtain the occupancy rates of data types within the data cache. This is both to identify if exploiting the per-bit vulnerability variation of NPB-serial in a data cache adheres to the high exploitability characteristics of Subsection 4.1.1 and also to help measure the optimal potential benefits in Section 4.3. The potential benefits will be shown in terms of strongly-protected surface reduction against to what effect on the fault coverage. These benefits are presented to demonstrate the upper exploitation limits of the application behavior under data corruption. To do so, we assumed an optimal fault-tolerant cache design that can adapt to unequally protect its contents. Therefore, before concluding this chapter, we discuss on the practicality of our exploitation approach and on ways to improve it.

4.1 Exploitation Alternatives

There are numerous alternative ways, design targets, areas and mechanisms (also shown in Fig. 4.1) that offer grounds for exploiting the application behavior under data corruption.

Out of all the possible ways and targets to exploit the observed characterization insight, we focus on exploiting it for shifting into **vulnerability-driven unequal-protection** architectures to **reduce** the reliability associated costs with a minimum effect on the fault coverage against **SDCs**. Out of all the gained insight, we focus on exploiting the **data-level** vulnerability variations and, specifically, the **per-bit** variations. Finally, out of all the areas we could apply the insight, we chose to apply it on a **hardware-level** protection mechanism that operates, specifically, in a **data cache**.

In this section we explain our reasoning for limiting our exploitation investigation into the aforementioned subset of the alternative exploitation options.

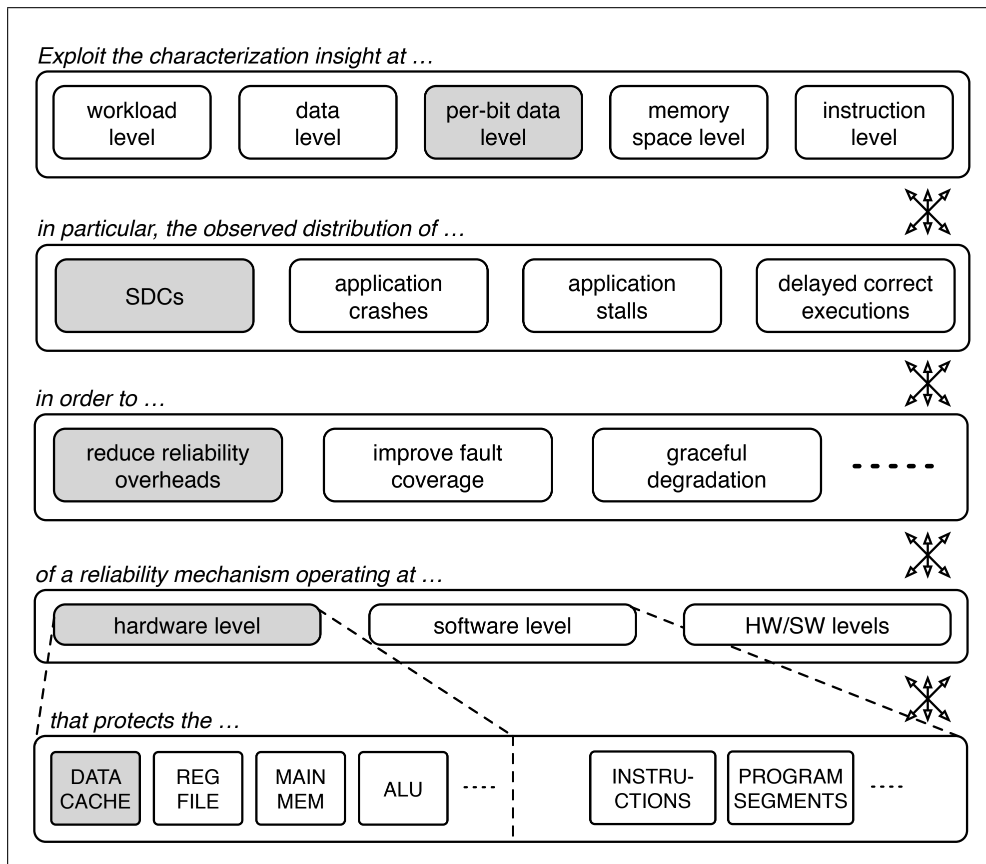

Figure 4.1: Visualization of the alternative options to exploit the application behavior under data corruption. Grayed boxes show our choice.

The characterization study in the previous chapter provided with detailed breakdowns for all possible application behavior outcomes under data corruption. Throughout this thesis, as the vulnerability factor of a hardware/software area we assumed the statistical probability that a hardware fault occurring in that area would cause an SDC. Therefore, we will focus on exploiting the variations of the reported **SDC** occurrences only. Nevertheless, one could also exploit the distribution of the rest corruption outcomes too, if the design target was to reduce their occurrences.

We opted to consider only the SDC distribution and protect only against SDCs because they are the most severe corruption outcome. SDCs are the only execution outcome that is undetectable online by any means and that does not provide any indication that something happened out of the ordinary. All other corruption outcomes (application crash, application stall, delayed correct execution) are observable and detectable by conventional methods (e.g., software-visible symptom-based fault detection [50, 27, 37, 13, 8] monitors for suspicious behaviors/symptoms, such as violating

likely program invariants, memory exceptions, cache misses, branch mispredictions, fatal exceptions, program crashes, high OS activity, hangs).

Generally, the focus has always been on eliminating the occurrence of SDCs only. Such related works that focus on increasing the dependability of systems consider protection against SDCs only and assume special detection mechanisms for the rest corruption outcomes. Moreover, the fault coverage of the proposed fault tolerant approaches is always experimentally evaluated in terms of the reported resulting SDC occurrences only.

In the rest of this chapter the characterization insight will be exploited focusing on areas less vulnerable to result into SDCs only.

The characterization study in the previous chapter essentially estimated vulnerability factors for different hardware/software constructs for given workloads. As the vulnerabilities varied they provided a way to rank these constructs based on their vulnerability and pointed out which areas are more or less likely to silently corrupt the output if a hardware fault occurs there.

What we propose here is that it is unnecessary to protect everything equally and exhaustively and, instead, we can exploit the vulnerability insight to our advantage by shifting to **vulnerability-driven unequal-protection** mechanisms where the protection strength is assigned to hardware/software parts according to their vulnerability.

In the rest of this chapter the vulnerability insight will be exploited for the purposes of shifting to vulnerability-driven unequal-protection mechanisms within a given structure or same-level structures.

It is intuitive how shifting to such a vulnerability-driven protection paradigm can be beneficial compared to protecting everything equally and exhaustively. Interestingly, a vulnerability-driven unequally-protected architecture may target different design objectives.

One promising way to exploit the vulnerability insight in a vulnerability-aware manner is for reducing the cost of existing exhaustive fault-tolerant mechanisms. Compared to an equally protected approach, we can **reduce the reliability overheads** in a given structure by avoiding its exhaustive unnecessary protection with a minimum impact on the total dependability by reducing the protection strength (and thus the cost of protection) of the less-vulnerable parts only of the given structure.

Targeting reliability cost reduction is one of the design targets that vulnerability

variations can be exploited for in vulnerability-driven protection mechanisms. An alternative design target could be to improve the fault coverage at the same reliability overhead as an equally-protected approach, by reducing the protection strength of the less-vulnerable parts while increasing the protection strength of the more-vulnerable parts. Such a design target is very relevant today for keeping up with the increasing number of error rates that make exhaustive schemes invoke unsustainable costs.

Furthermore, vulnerability-driven protection could be used for trade-off purposes. As our proposed class of vulnerability-aware fault-tolerant architectures can have varying fault-protection levels, it lends itself naturally to offer various degrees of reliability QoS and trade off reliability against performance/cost. Similarly, it shows promise for a gracefully degrading system that can still operate under a large number of errors while offering a necessary (but reduced) set of services.

The characterization insight is not necessarily to be exploited for design-time purposes. As mentioned throughout the previous chapter, there are various other areas of exploiting the vulnerability variations. Alternative uses could be for reducing the testing space during application dependability assessment by avoiding testing the less-vulnerable application data, for reducing the testing space during reliability mechanism dependability assessment, for guiding a reliability-aware software transformation, etc.

In the rest of this chapter the vulnerability insight will be exploited for the purposes of reducing the cost of the reliability overheads within a given structure or same-level structures.

One of the key features of our characterization study was that it operated at application data level to observe the vulnerability variation between different data types (according to their usage type). Contrary to existing works that investigated the vulnerability of higher-level hardware structures and could only exploit their observations only for the structures under consideration, we were able to investigate on the vulnerability variations of more areas than the one we injected faults in. As shown throughout the previous chapter, although our characterization framework operated on results by *data-level* fault-injection experiments only, its data awareness allowed to extrapolate the reported results to model the vulnerability of other areas too without the need to remodel and repeat the fault-injection tests. As a result we were able to obtain the vulnerability variations within the memory space, the register file and among instruction-level characteristics.

Despite that, out of all the gained insight, we chose to exploit the application data

related vulnerability variations (Section 3.3) and, in particular, the per-bit ones (Section 3.3.1). As our study has been data-level aware it is more natural to focus on exploiting the application data vulnerabilities. More importantly, the **per-bit data-level variations** show clear and consistent vulnerability variation patterns for given combinations of data usage types and sizes; distinct bit ranges within application data can be identified with distinct different vulnerability levels.

As this points out that application data are vulnerable in parts, it offers straightforward grounds to exploit that by protecting unequally the application data parts. Moreover as this goes down to bit-level vulnerability estimation it can fine-tune the unequal protection so that the potential benefits are maximized and, considering that our suggested vulnerability-driven unequal-protection shift can offer various degrees of reliability QoS, it can offer an even smoother trading off against reliability.

Most importantly, the per-bit data-related vulnerabilities show *high-exploitability characteristics*. There are clear patterns, with high degrees of variation, that are consistent among most applications. Moreover it is easy to unequally protect data words (both in terms of identifying their types to drive the unequal protection and in terms of implementing the unequal protection on them).

The rest observed vulnerability variations tended to not offer such clear insight among different benchmarks. Nevertheless there is still ground for exploiting them in a per application basis, as long as first the respective vulnerabilities are ranked before exploiting them.

In the rest of this chapter only the per-bit data-type related vulnerability variations will be exploited.

Depending on the level where error detection and error correction take place, reliability mechanisms can be classified into hardware level, software level or a combination of both. As vulnerability-driven unequal-protection is orthogonal to the specifics of a reliability mechanism, the vulnerability insight can be exploited at any-level mechanisms. E.g., a software-level mechanism that ensures reliability by instruction duplication can be adapted to duplicate only the instructions processing the more-vulnerable data. Similarly, a hardware-level mechanism that ensures reliability by replicating hardware structures can avoid duplicating parts of those that process less-vulnerable data.

Equally interesting is applying vulnerability awareness to hardware/software co-design reliability approaches. Such a shift offers more space for trading-off reliability

QoS against performance/cost. Both the hardware and the software layer can become aware of the error rates and exchange relevant information to ensure that the system operation will always move forward in a gracefully degrading manner. In such co-design approaches the hardware layer knowingly may not provide the illusion of fault-free operation and let some faults propagate to the software layer, where the software would be fault-aware to correct these in a way it sees fit.

Despite the choices made so far, vulnerability-driven unequal-protection may be introduced to reduce the overhead costs of reliability mechanisms operating at *any* level and protecting *any* area. In other words, the characterization insight can be applied orthogonally and exploited in many hardware/software areas. As we specifically decided to exploit the per-bit data-level vulnerability variations, it is more fitting to do so in the case of a **hardware-level** protection mechanism operating in a **data cache**.

Out of the possible hardware areas that require protection against faults, this chosen insight is more natural to be exploited in data-holding structures in general (i.e., main memory, data caches, registers, etc.) and not necessarily only in data caches. Nevertheless, as mentioned before, due to the data-level awareness of the experiments, our insight can be extrapolated to be applied in other non data-holding structures (i.e., arithmetic/logical operations, communication wires, etc.). Moreover, data-holding structures are more susceptible to transient hardware faults due to their increasing relative size and due to their constant exposure to faults compared to logic that is exposed only when it is actively computing a result [29]. As expected the extra cost of constant full protection is also relatively higher and along with data caches being a performance bottleneck, research on decreasing the data cache reliability overhead is more relevant.

Equally important is that exploiting the per-bit data-level vulnerabilities shows more *high exploitability potential* in data caches than other data-holding structures. Exploiting that application data are vulnerable in parts depending on their usage type is more straightforward in the case of an vulnerability-aware unequally-protected data cache compared to a similar main memory. Determining the data types as they are being accessed and assigning their vulnerabilities in a data cache is easier, compared to the main memory, where data reside before being first used and thus before identifying their types. Similarly, if we were to exploit the data-level vulnerabilities in an unequally-protected register file, the potential savings will be relatively small as the less vulnerable data wouldn't occupy specific registers nor a large set of them constantly. Moreover, as it will be established in the next section, the most consistent per-bit insight for the tested applications (the clear vulnerability variations among FP

data) is highly exploitable in a data cache as it is occupied by it in large volumes during large application execution intervals.

In the rest of this chapter the vulnerability insight will be exploited only for the case of a hardware-level mechanism protecting a data cache.

4.1.1 Identifying Areas of High Exploitability Potential

Throughout this section a variety of alternatives to exploit the vulnerability insight was discussed. The abundance of options is mostly enabled by the fact that (a) we obtained vulnerability variations at different levels and (b) that unequal protection can be applied orthogonally to many areas regardless of the specifics of the original reliability mechanism (i.e., their protection method, the component they protect, their level of operation or their design goal).

All exploitation alternatives do not share the same magnitude of potential benefits. Here we suggest what characteristics are desired to *identify promising areas of high exploitability potential to maximize the benefits of exploiting the application behavior insight by introducing vulnerability-aware unequal protection* and how our chosen exploitation alternative satisfies these properties.

Generally, the exploitability potential depends on which observed vulnerability insight is to be exploited and where the vulnerability-aware unequal protection is to be introduced. To maximize the potential benefits the following characteristics are desired:

- Regarding the vulnerability insight to be exploited, it is desired that it offers:
  (a) clear and distinct vulnerability patterns,
  (b) that are consistent across different workloads, and
  (c) that their vulnerability intensity levels are different by high degrees. This is because high vulnerability variation between the less and more vulnerable areas offers more potential benefits compared to constant vulnerability throughout all areas.
- Regarding the location where the vulnerability-aware unequal protection is to be introduced, it is desired that:
  (a) it is easy and low cost to assign the protection strength based on the vulnerabilities; in other words, identifying the vulnerabilities of the different parts must be easy and low cost to occur online during execution, and

  (b) it must be possible to introduce unequal protection at the area under protection without much overhead; in other words, the existing mechanisms must be naturally adaptable to unequal protection.

- Regarding the synergy of both of the above, for a vulnerability pattern under exploitation in the chosen area, the volume/timeshare that the varying vulnerability pattern is present in the chosen area must be large enough to maximize the benefits and to warrant the effort of introducing vulnerability-aware unequal protection there.

Out of all the available vulnerability insight and potential exploitation areas, we decided to exploit the *per-bit data-level vulnerability variations of the NPB-serial benchmarks* to reduce the reliability-overhead costs in a *reduced-cost vulnerability-driven unequally-protected data cache*. This chosen exploitation alternative shows all high exploitability characteristics.

In particular, regarding the per-bit data-level vulnerability insight for NPB-serial (Section 3.3.1): (a) It showed clear and distinct vulnerability patterns among different bit ranges of most data types, where the bit ranges that are less vulnerable are either at the MSBs or LSBs of application data. (b) It showed consistent vulnerability variation among most of the different workloads. (c) The observed vulnerability patterns have different vulnerability intensity levels; the less-vulnerable bit ranges have almost zero vulnerability factors, while the more-vulnerable are significantly more vulnerable.

Regarding introducing vulnerability-aware unequal protection in a data cache: (a) As the per-bit vulnerabilities are determined by the data type and as most data types can be identified at access time, it is possible to set the vulnerabilities of the cache contents at access time during execution to drive the vulnerability-aware mechanism online. (b) As the data cache is a data-holding structure, the available data protection mechanisms (e.g., ECC) can naturally be adapted to support unequal data word protection.

Regarding the volume/timeshare that the most exploitable per-bit data-level vulnerability patterns occupy the data cache: The most promising per-bit data-level vulnerability pattern is the one for FP data because they show clear, consistent and distinct patterns. Due to the FP-heavy nature of the NPB-serial benchmarks, we expect that the data cache will be heavily occupied by FP data during large application execution intervals. Therefore there is high exploitation potential in unequally protecting FP data in a data cache. Although we expect that not all data types will experience high volumes in the data cache (and thus not experience all high exploitability characteristics), those that will are looking very promising, even if they are the only ones to be exploited.

Generally, we expect that the potential benefits will be substantial and can be maximized when unequally protecting application data in a data cache.

As the per-bit data-level vulnerability insight for NPB-serial showed, there are wide and continuous bit ranges within specific application data with near-zero probability to result into an SDC. E.g., each of the 32 LSBs of FPs in CG have less than 1% probability to cause an SDC if corrupted. Similar near-zero probability is experienced for up to 49 MSBs of 8-byte IPs, for up to 41 MSBs of 8-byte PTRs, for up to 39 MSBs of 8-byte PTRMRs and PTRTPs (excluding CG), etc.

This points out that only parts of data need to be protected strongly against faults. E.g., only 50% of the full width of FPs in CG, 24% of 8-byte IPs, 36% of 8-byte PTRs, 40% of 8-byte PTRMRs and PTRTPs (excluding CG) needs to be protected strongly against faults. If the rest non-vulnerable bit ranges are not as strongly protected as the more vulnerable ones, the expected fault coverage per dataword of these types will almost not drop at all.

When taking into account the volume of the data types in a data cache, generally we expect that the strongly protected surface reduction can be maximized when exploiting the per-bit data-level vulnerabilities. E.g., as NPB-serial is FP heavy, we expect that the data cache will be largely occupied by FPs. Assuming an average occupancy rate of X% of FPs in a data cache running CG, then $0.5 * X\%$ of the data cache can be protected not-strongly to an almost zero drop of the fault coverage of FPs. Therefore we can reduce the total strongly-protected surface to a minimum effect on the fault coverage to reduce the reliability overheads in a cache.

The potential benefits of our chosen exploitation alternative depend on the volume that the data types occupy the data cache. In the next section we will profile the contents of a data cache running the NPB-serial benchmarks in order to obtain the volume/timeshare of data types within the data cache. This will further show how the chosen exploitation alternative experiences all high exploitability characteristics and how this can translate into potential benefits in terms of strongly-protected surface reduction.

4.2 Data Cache Content Profiling

To move forward with our exploitation investigation we require a method for *data cache content profiling* to obtain the volume/timeshare of *data types* within a *data cache* during the execution of an application. Such information is needed (a) to reaffirm that our chosen exploitation alternative (of per-bit data-level vulnerabilities in a data cache) experiences all high exploitability characteristics (see previous section) and (b) to measure the maximum potential benefits of our chosen exploitation alternative (see next section).

In this section, first, we describe our data cache content profiler. Its purpose is to simulate a data cache in order to obtain the occupancy rates of the data types within the data cache during the execution of an application. Then, after setting up our experiments, we profile the contents of various data cache configurations executing the NPB-serial benchmarks to obtain the required information.

4.2.1 Data Cache Content Profiler

Overview: For a given application binary and a cache configuration, the data cache content profiler instruments the application to capture every memory access (both loads and stores) and to simulate them in the data cache. The cache content profiler requires one instrumented run of the application for each different cache configuration.

For every memory access, it classifies the accessed data in different usage types, according to their use by the application. The usage type classification logic is the same as the one used by the single-fault injection tool[1].

Then, all *classified* memory accesses are simulated *in order* by a data cache simulator. On top of traditional cache simulators, it keeps track of the data type that each of the cache bytes is occupied by. This enables to obtain the occupancy rates of the data types within the data cache during the execution of the given application.

Data usage type classification: Classifying the usage type of the accessed data can be either immediate at access time or it may require tracking the data through the execution until a first meaningful use (as detailed in Subsection 2.2.3). As the cache content profiler requires one run of the application, there may be multiple accessed

[1]Given the reasons we require data cache content profiling, the profiler has to be compatible with the performed vulnerability characterization, thus it must classify the contents using the same classification algorithm as the characterization framework (see Subsection 2.2.3 and Fig. 2.2).

data being tracked concurrently but at separate tracking processes.

To manage the multiple data tracking processes we use a *queue*. This queue keeps an *in order* history of the memory accesses and may include both classified and not-yet classified memory accesses. Each element in the queue is independent of each other. There are separate data structures used for implementing the independent data tracking and dynamic taint tracking logic; each queue entry holds all the necessary propagation information for its own purposes, i.e., load/store, initial memory address, initial register, tainted registers, tainted memory locations, instruction tracking window, etc.

Once a new memory access is captured, it is added to the queue. If its data usage type can be classified immediately, it is flagged accordingly. Otherwise, the classification process is set up and all relevant information for data tracking purposes are added to the new queue element. If it is a load access, this includes the original loaded memory location and the first register to store the loaded data. If it is a store access, this includes only the original stored memory location.

Then, for every instrumented instruction and for every non-classified memory access data type, the classification proceeds independently of the rest ongoing classifications processes; the data tracking and/or the dynamic taint tracking proceeds for each memory access according to the classification logic. Once a memory access data type is classified, the relevant queue entry is updated.

The logic and process to detect the types by data tracking is the same as in the single-fault injection tool. A separate case holds for classifying the usage types of data accessed by memory stores. When storing data in memory there is no indication how the data have been used so far by the application. An exception holds if they were stored in memory using one of the special registers as an input register and then we can classify them immediately, e.g., a store using an FP register as an input can be classified as accessing FP data. Otherwise, to classify the originally stored data, we will have to wait for them to be accessed again by a subsequent memory load access and then invoke the classification process.

Queue management: To ensure the correctness of the forthcoming simulation we use the aforementioned *queue*. To simulate a memory access by the cache simulator, its usage data type must have been classified. Also, the memory accesses must be simulated in the correct order. The queue, apart from enabling the concurrent data tracking processes, keeps an *in order* history of the memory accesses to feed them *in order* to the cache simulator.

As data usage type classification is not always immediate and does not take the same time to detect, the queue may include both classified and not-yet classified memory access data types; some elements in the queue may not be classified yet but more recent ones may be.

For every instrumented instruction we attempt to empty the queue. All classified memory accesses at the head of the queue are removed from the queue, until the head of the queue is occupied by a not-yet classified memory access. Those that are successfully removed from the head of the queue are passed to the cache simulator to simulate a *memory access* (load or store) at a *memory address* that is of a specific *size in bytes* and that holds data of a classified *usage type*.

If the head of the queue remains unclassified for large execution intervals, the queue size may quickly become unmanageable. To avoid an explosion of the queue size, we impose (a) a time limit (in instructions) for the classification process by data tracking[2], and (b) a time limit (in instructions) for waiting for a non-classified memory store location to be accessed again by a memory load[3].

Fig. 4.2 shows an example of how the queue is populated over time to perform concurrent different data usage type classification processes on different memory accesses and how the queue is emptied to pass the classified memory accesses to the cache simulator.

Data cache simulator: The classified memory accesses that are removed from the head of the queue are passed in order to the cache simulator. The passed information include the access type (load or store), the memory access address, the accessed data size (in bytes) and the accessed data usage type.

The simulator is a traditional cache simulator that, on top of the usual ones, it keeps track of the cache content data types at byte granularity. For every cache byte, it stores the usage type of the data that occupy that byte and the full byte range that the data in that byte occupy. E.g., assuming an 8-byte FP stored in memory at bytes 0x10 up

[2]As in the single-fault injection tool, an instruction window of 1000 instructions is set as a time limit to classify the tracked data's usage type. If the time limit is reached, their usage type is reported as INT. Due to this time limit, data that may have otherwise classified as other usage types, are forced to be classified as INT. An implication of that is that the percentage of INT data may increase and include a mix of other types too, while the rest reported data usage types contain exactly the correctly identified types. Nevertheless, the exploitation potential benefits presented in the rest of this chapter still hold, especially as this classification behavior was present also during the vulnerability characterization framework.

[3]Memory stores that are not immediately classified cannot be tracked to determine their usage type until accessed again by a subsequent memory load. If this doesn't occur within an instruction window of 1000 instructions, then the store access is reported as UNUSEDSTORE.

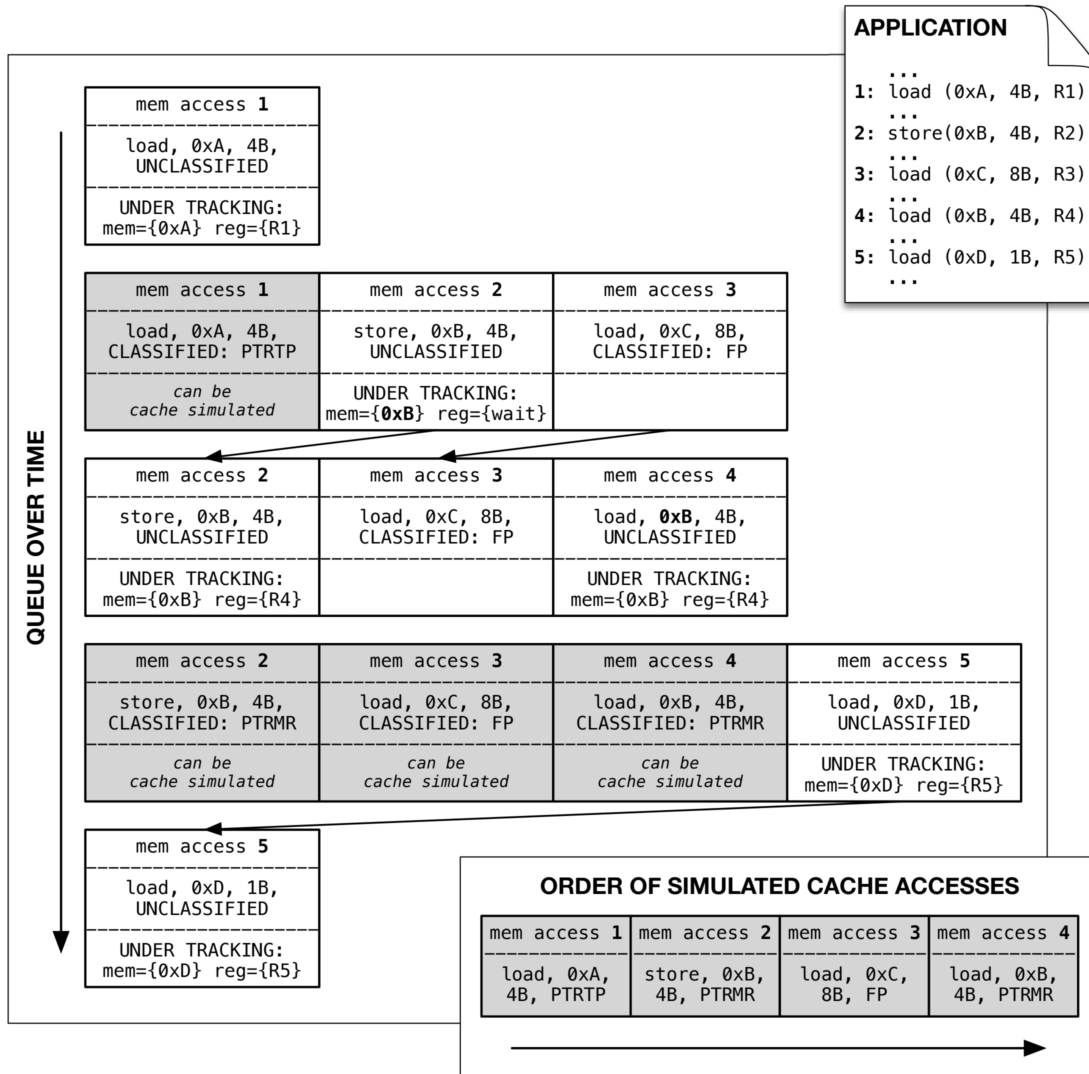

Figure 4.2: Example management of concurrent usage type classification for in-order data cache content profiling.

(A) Each memory access is added to the queue. If classified immediately, it is flagged accordingly. Otherwise, the necessary information to start the data tracking are added. Memory access 1 requires data tracking, thus a classification process is setup to start on the accessed memory address (0xA) and the first register loaded (R1). Memory address 2 is a store access and will have to wait until the accessed memory address (0xB) is loaded by a subsequent access (memory access 4), thus the classification process waits until memory access 4. Memory access 3 is loaded into an FP register and can be classified immediately.

(B) Over time all unclassified accesses go through their respective classification process. The figure shows the changes over time and not per individual instrumented instruction. When the head of the queue is populated by one or more classified accesses, these are removed and passed in order as cache accesses to the type-aware cache simulator.

to 0x17 and loaded into the cache, all simulated cache bytes representing 0x10 up to 0x17 are marked as storing an FP that spans from 0x10 up to 0x17.

For every cache access (both hits and misses), the exact cache bytes that are accessed are always updated to the usage type of the newly accessed data, even in the case of load hits. As we care about usage type profiling, the cache content type information must remain always up to date and reflect the most recent usage of the data. E.g., assuming a cache hit of an 8-byte IP stored in memory at bytes 0x10 up to 0x17, even though it is a cache hit, all simulated cache bytes representing 0x10 up to 0x17 are changed (from the aforementioned FP) to be marked as storing an IP that spans from 0x10 up to 0x17.

On a similar note, when a cache access causes the change of a cache byte usage type, all neighboring bytes that were part of the old value must have their usage type invalidated. E.g., assuming a cache hit of a 1-byte INT stored in memory at 0x14, the simulated cache byte representing 0x14 is changed (from the aforementioned IP data stored between 0x10 and 0x17) to be marked as storing a 1-byte INT and (because the original IP data are no longer valid) the type of the cache bytes representing 0x10-0x13 and 0x15-0x17 is invalidated.

Finally, due to classifying data types according to their use by the application, the simulated usage types are known only after the data are used. Therefore, in cache misses that cause a new cache line to be loaded into the cache, we have no way of knowing the data types of the full cache line, apart from the bytes that were actively accessed by the application. As a result, the profiled cache contents show a live image of the types accessed so far by the application and not an indication of future accessed types.

Reported occupancy rates: Every time a cache byte type is changed there are relevant counters that are updated accordingly. These counter provide a live image of exactly how many cache bytes are occupied by any combination of usage types and sizes. The data cache content profiler uses these counters to report the occupancy rates of the data types within the data cache during the execution of the given application.

4.2.2 Experimental Setup and Profiling Results

The data cache content profiler was implemented as a set of dynamic binary instrumentation Pin tools [28] reusing the usage type classification logic of the single-fault

injection tool (as detailed in Subsection 2.2.3). Given the reasons behind the data cache content profiling, the profiling results must be compatible to the performed vulnerability characterization. Therefore we profiled the data cache contents when executing the same benchmark binaries as the ones we have already obtained the vulnerability insight for.

The contents of twelve different-sized cache configurations were profiled while executing the full set of the ten workloads of the NAS Parallel Benchmarks [10] (64-bit, NPB-serial, version 3.3.1, input class size S, gcc 4.4.6 -o3, Linux kernel 2.6.32). The data caches were 4-set associative with a 32-byte block size and their size varied from 1K up to 2048K. To perform the cache content profiling one run of each application was instrumented for each different cache size.

Given the data cache content profiler design, the results effectively depict the data occupancy rates in an L1 cache. Focusing on L1 caches and reducing the reliability costs at that level is more imperative in terms of cost and performance than in lower level caches. Although we profiled up to 2048K-sized caches, that are unreasonable to be used as L1 caches, these results still give an insight on how the larger lower level caches would be occupied by data types.

Fig. 4.3 shows the data cache content profiling results for the CG benchmark for each profiled cache size. It can be seen that, throughout the execution, there are data types (FP and PTRTP data in the case of CG) that dominate the data cache in large volumes.

Similar behavior is exhibited in the rest tested benchmarks[4]; there are data types that occupy the data cache in high volumes and during large execution intervals. This holds especially true for the case of FP data. For most of the rest tested benchmarks, except DC and IS that are not FP heavy, there are high occupancy rates of FP data in the data cache (irrespective of its size) throughout the execution.

What varies among the profiled data caches and benchmarks is the magnitude of the occupancy rates. Fig. 4.4 shows the breakdown of the data cache contents according to their usage type for all NPB-serial benchmarks. These results are averaged occupancy rates exhibited in each profiled cache size over the full execution of each benchmark. This averaged breakdown of occupancy rates reaffirms that there are data types (especially the FP data) that reside in large volumes in the data cache, regardless of the cache size, during the lifetime of most profiled applications. As these are the

[4]For space reasons, the profiled data cache contents over the execution time are shown only for one of the ten profiled NPB-serial benchmarks.

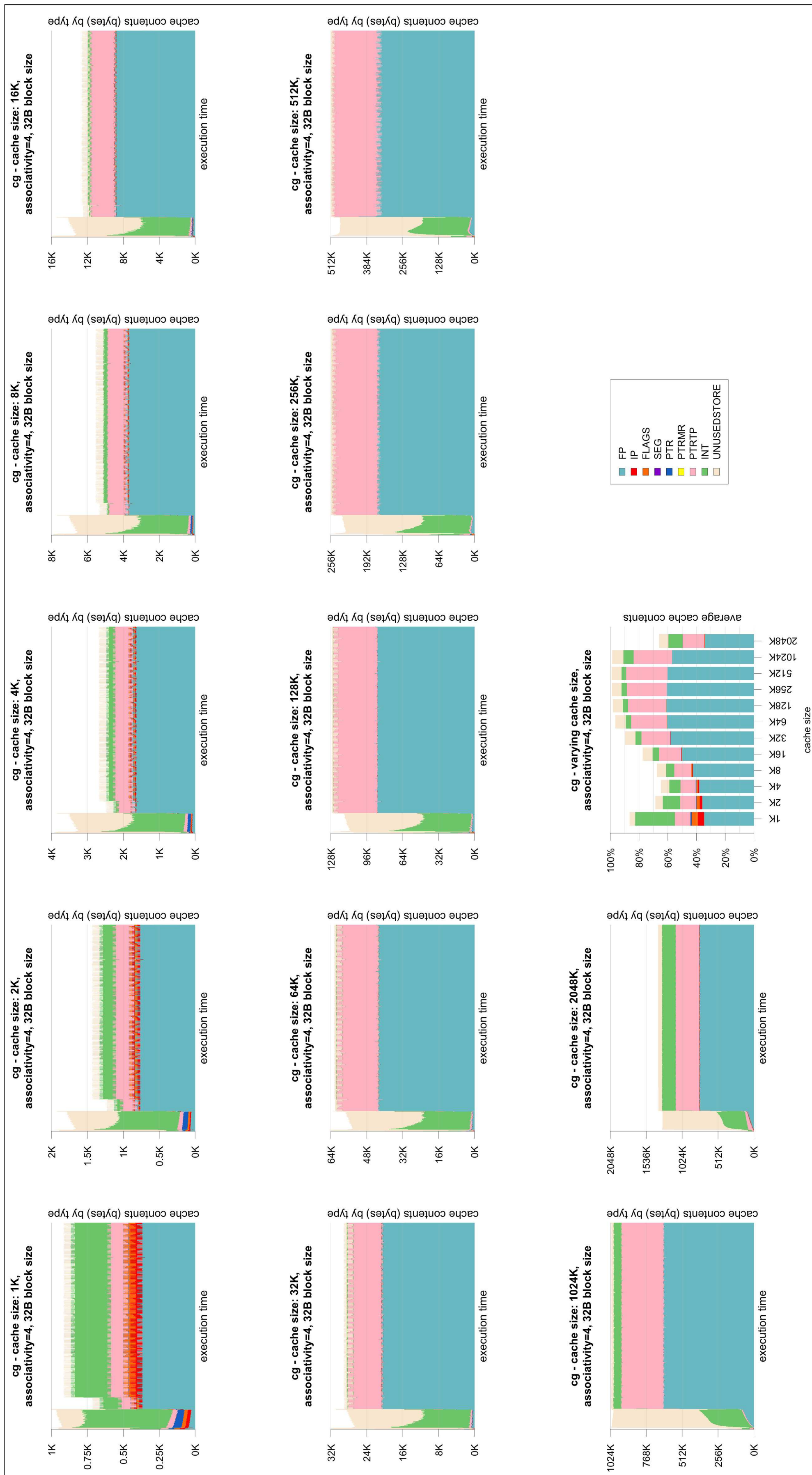

Figure 4.3: Breakdown of data cache contents (according to their usage type) over the execution of the CG benchmark and on average (last plot). (Varying cache size: 1K, ..., 2048K, 4-set associativity, 32-byte block size)

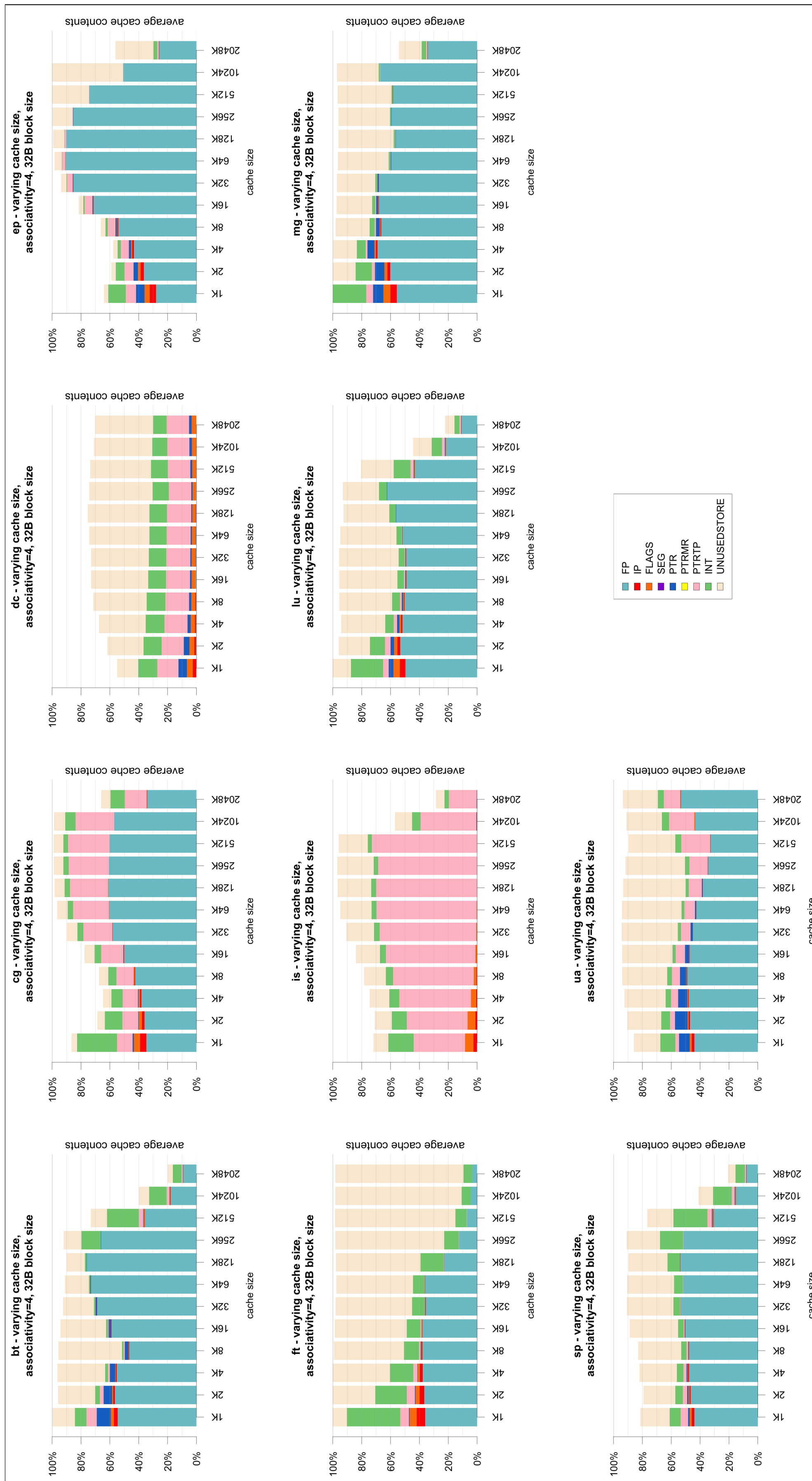

Figure 4.4: Averaged breakdown of data cache contents (according to their usage type) during execution of all NPB-serial benchmarks. (Varying cache size: 1K, ..., 2048K, 4-set associativity, 32-byte block size)

data types that we intend to exploit their vulnerabilities, the profiling results motivate that the potential benefits will be substantial.

As it appears that our chosen exploitation alternative satisfies all high exploitability characteristics, we can move forward on measuring the maximum potential benefits. In the next section we use the obtained occupancy rates to estimate the potential benefits of exploiting the per-bit data-level vulnerability variations of NPB-serial in a vulnerability-aware unequally-protected data cache.

4.3 Per-Bit Data-Level Vulnerability Exploitation Benefits in a Data Cache

In this section we show the maximum potential benefits of our chosen exploitation alternative. In particular, we will exploit the *per-bit data-level vulnerability variations of NPB-serial* targeting the *reliability-overheads reduction* in a *vulnerability-aware unequally-protected data cache* that offers two levels of fault-protection strength while running the NPB-serial benchmarks.

As mentioned throughout this chapter and further established by the data cache content profiling, we expect that exploiting the characterization insight in that particular vulnerability-driven manner will have significant potential cost-saving benefits, especially as it indeed experiences all high-exploitability characteristics (Section 4.1.1).

The potential reliability-overhead benefits will be measured in terms of strongly-protected surface reduction against to what effect on the fault coverage to enable trading-off the reliability overheads against the reliability QoS level.

To estimate these potential benefits (for the case of the data cache running the NPB-serial) we only require (a) the observed per-bit data-level vulnerability variations of the NPB-serial benchmarks obtained during the characterization study (Section 3.3.1) and (b) the observed occupancy rates of data types in a cache executing the NPB-serial benchmarks obtained by the data cache content profiling (Section 4.2). As explained later, this limit study does not require an exact design for a vulnerability-aware unequally-protected cache; an assumed optimal cache design suffices.

As we exploit the per-bit *data-level* vulnerability insight, the trade-offs will be investigated at application data level granularity per particular combinations of data usage

types (D) and data sizes (B bytes, b bits), while fine-tuning their unequal protection at bit granularity.

More precisely, we will introduce the *unequal protection per particular combinations of D-B data*, where their full bit range will be protected against N hardware faults and $b-l$ bits will be protected stronger against *one extra* hardware fault. This enables to show how varying the amount of bits under stronger protection ($b-l$, $l \in [0,b]$) affects the fault coverage[5] of D-B data against $N+1$ faults occurring randomly over the b bits of D-B data. Meanwhile, we will show how the variations of l for the particular combination of D-B data translates to benefits on the *full* data cache in terms of strongly-protected surface reduction.

Therefore, the trade-offs we observe are between the fault coverage *of particular D-B data* against the potential benefits *in the full cache*. Investigating the effects on fault coverage at application data level (per combinations of D-B data) and not in higher granularities (e.g., cache line, full cache), makes it clear to see how varying the protection of D-B data affects their fault coverage to what benefits to the full cache.

If we had investigated the fault coverage variations at a higher level, although (a) the unequal protection would still be per D-B data and (b) the benefits would still be the same, the effects of exploiting the D-B data vulnerabilities (by varying the unequal protection) on fault coverage would not be seen clearly. This is mainly due to the methods for measuring the theoretical fault coverage that involve the probability that a single hardware fault occurs in one out of the many cache bits. Due to the cache sizes, this probability is small enough that makes the fault coverage changes minimal and hard to see.

The potential benefits will be estimated for the case of a 64K data cache (4-set associative, 32B block size). Out of the 12 profiled cache configurations (Section 4.2) we show the exploitation potential in the 64K data cache only, as it is a typical size for an L1 cache where any improvements in cost and performance are more imperative than lower level caches. Moreover, the 64K-sized configuration experienced the most instances of highest occupancy rates of FP data over the profiled benchmarks (Fig. 4.4). Nevertheless, despite selecting the 64K-sized configuration, similar analysis can be performed for the rest cache configurations.

Table 4.1 shows the data occupancy rates in a 64K sized data cache running the

[5]As the **fault coverage** (of an area protected by a reliability mechanism against a number of occurring hardware faults) we define the probability that, if the number of faults occurs in this given area, an SDC will not occur.

Table 4.1: Average occupancy rates (as percentages of the total cache area) per profiled benchmark for different combinations of data usage types and data sizes that reside in a 64K data cache executing NPB-serial. Rates obtained by the data cache content profiler (Section 4.2).

AVERAGE OCCUPANCY RATES (%) IN A 64K DATA CACHE (4-SET ASSOCIATIVE, 32B BLOCK SIZE) EXECUTING NPB-SERIAL (3.3.1, INPUT CLASS S)																		
type:	FP			IP	PTR				PTRTP			INT			UnusedStore			
size:	8	16	any	8	4	8	16	any	4	8	any	4	8	any	4	8	16	any
BT	72.95	0.02	72.97	0.09	0.01	0.35	0.02	0.38	0.00	0.18	0.18	0.10	0.73	0.83	0.00	16.58	0.02	16.60
CG	60.42	0.03	60.45	0.08	0.00	0.03	0.00	0.03	24.65	0.11	24.76	3.45	0.34	3.81	0.91	4.51	1.79	7.21
DC	0.00	0.63	0.63	0.09	0.02	0.77	0.00	0.79	0.00	16.61	16.62	1.10	10.18	11.99	0.59	37.85	1.37	41.70
EP	90.64	0.02	90.66	0.07	0.00	0.10	0.00	0.10	0.00	2.10	2.10	0.08	0.11	0.19	0.00	4.93	0.01	4.94
FT	36.04	0.00	36.04	0.10	0.01	0.06	0.00	0.07	0.00	0.22	0.22	0.22	7.77	7.99	0.00	53.07	0.00	53.07
IS	0.00	0.00	0.00	0.04	0.00	0.02	0.00	0.02	69.18	0.10	69.28	3.07	0.38	3.50	20.93	0.12	0.65	21.70
LU	51.54	0.04	51.58	0.07	0.01	0.15	0.00	0.16	0.02	0.25	0.27	0.19	3.41	3.62	0.00	39.03	0.01	39.04
MG	59.46	0.00	59.46	0.07	0.00	0.28	0.14	0.42	0.01	0.21	0.22	0.20	0.84	1.07	0.01	31.76	3.34	35.11
SP	51.71	0.01	51.72	0.04	0.01	0.06	0.00	0.07	0.01	0.14	0.15	0.09	5.82	5.91	0.00	32.41	0.07	32.48
UA	42.41	0.03	42.44	0.05	0.00	0.80	0.00	0.80	7.37	0.05	7.42	0.71	1.35	2.06	0.11	34.36	6.71	41.18

NPB-serial benchmarks, as obtained from the data cache content profiler. Each rate shows the average percentage of the total cache area that was occupied by a specific data type of a particular size[6]. Not all cache occupancy rates for any combination of data types and sizes are shown. Although the occupancy rates were obtained for all combinations, only a subset of those will be exploited here as many of them were very low for significant benefits.

This limit study does not require an exact data cache design to measure the intended potential benefits. It only needs the per-bit data-level vulnerability variations and the data cache occupancy rates. Therefore, we can assume an available optimal *vulnerability-aware unequally-protected data cache* design that (a) can offer two levels of fault-protection strength that can be fine-tuned at bit granularity to unequally protect the cache contents at data-level granularity and (b) can detect the vulnerability of cache data to drive their unequal protection according to their expected vulnerabilities. Although assuming an optimal cache design assists into estimating the maximum potential benefits of introducing vulnerability-aware unequal protection in a data cache, it is

[6]Note that during the data cache content profiling (Section 4.2) we showed the total occupancy rates per data type of any size. Here we will use the occupancy rates per data type for particular data sizes to ensure compatibility with the respective per-bit vulnerability observations to estimate correctly the trade-offs.

an indication too of the limited practicality of the proposed approach as we will discuss in the end of this chapter.

4.3.1 Strongly-Protected Surface Trade-Offs Against Fault Coverage

In this subsection we demonstrate the potential exploitability of our characterization findings in a generic design of a fault-tolerant data cache running the NPB-serial benchmarks. Assuming a vulnerability-aware unequal protection mechanism we answer the question of how much we can exploit the vulnerability characteristics of application data to reduce the strongly-protected surface of a data cache and to what effect on the reliability QoS level.

Introduction: The characterization study (Chapter 3) uncovered a variety of vulnerability characterization observations that we could exploit in our case of an unequally-protected data cache. Starting at higher application-level granularity, we showed the percentages of tests per application that led to SDCs (Fig. 3.1) and we can exploit this in a naive way by strongly protecting only that percentage of data cache contents, when the respective application is executing, agnostic to their independent vulnerabilities.

This, however, would not be effective without knowing how to match the less-protected portion of the cache to the less-vulnerable portion of the application's data. Instead unequal protection driven by the finer vulnerability classification of application data according to their attributes (Fig. 3.2-3.5) is more promising and accurate given that we can rank application data per application according to their vulnerabilities and protect stronger only the more-vulnerable data types in the data cache.

More interesting room for exploitation is in the per-bit vulnerability variations within application data (Fig. 3.7-3.14) as these exhibit the high-exploitability characteristics. Moreover, as these results can drive a bit-level finer-grained unequal protection, they allow for the maximum possible reduction of the strongly-protected surface to the minimum possible impact on the fault coverage.

Assumed cache design, protection and operation: Before measuring the fault coverage against the strongly-protected surface reduction, we assume a data cache that is equipped with a protection mechanism that can assign different protection at different cache contents at data granularity. This protection mechanism can offer two levels of

fault-protection strength that can be fine-tuned at bit-level granularity to protect the application data bits unequally. All bits of application data are protected against N faults (weaker protection strength), while some bits of some application data can be protected against one *extra* fault (stronger protection strength). Because our obtained vulnerability insight is for single-bit data corruption effects, in order to show the exploitability potential, we had to define the stronger protection as protecting against *one* extra hardware fault compared to the weaker protection.

The extra protected bits are protected by the stronger-protection level and the total of them constitutes what we call strongly-protected cache surface. When the strongly-protected surface covers the full cache, this narrows down to the fully equally protected data cache that protects every application data against $N+1$ faults and that we compare against. When there is no strongly-protected surface, this narrows down to a fully weakly-protected data cache that protects every application data against N faults and that exhibits the maximum potential benefits as it assumes that all contents are less vulnerable. When $N = 0$, the weakly-protected surface does not protect the area it covers.

In this trade-off analysis in terms of strongly-protected surface reduction, the exact value of N is irrelevant. The trade-off observations still hold for the case of stronger unequal protection against any number of N faults per protected application data. This motivates a strongly-protected surface reduction even when the more-vulnerable data are protected even stronger (or even have more than two protection-strength levels).

Similarly, the details on how exactly the protection mechanism is implemented are irrelevant for measuring the strongly-protected surface reduction. All we require is that the assumed protection mechanism can support the aforementioned behavior. Moreover, we assume the data cache has the mechanisms to detect the vulnerability of its data to drive their unequal protection accordingly. Once more, to measure the intended potential benefits, the related design choices and implementation details are irrelevant. E.g., how the vulnerability of data is detected, how the mapping is done of vulnerable data to stronger protected regions, how the cache is partitioned in two parts with different protection levels and to what effect on the other cache performance metrics, such as cache hit/miss rates, etc.

Fault coverage: As the fault coverage (of an area protected by a reliability mechanism against a number of occurring hardware faults) we define the probability that, if the number of faults occurs in the given area, an SDC will not occur.

To formulate this in the case of the assumed unequal-protection mechanism: The fault coverage $\mathbf{FC_{D,B,L}}$ (of a data type D of size B bytes when all of its b bits are protected against N faults and $b-l$ of its bits are protected against one extra hardware fault, where L is a set of $l \in [0,b]$ bits) is the probability that, if $N+1$ faults occur within the D-B data's b bits, an SDC will not occur.

Because all b bits are protected against N faults, the first N faults will not cause an SDC. As each occurring fault is an independent event, $FC_{D,B,L}$ narrows down to the probability that the $N+1$ fault will not cause an SDC. This is equivalent to the inverse probability of the $N+1$ fault causing an SDC. The probability of the $N+1$ fault causing an SDC is the sum of the probabilities that the $N+1$ fault occurs in a not strongly-protected bit and resulting into an SDC. Thus:

$$FC_{D,B,L} = 1 - \sum_{i \in L} \left(\frac{1}{b} VF_{D,B,i}\right) \tag{4.1}$$

The $\mathbf{VF_{D,B,i}}$ is the observed statistical probability that a corruption in the bit $i \in [0, b-1]$ of data of usage type D and size B bytes (b bits) will result into an SDC. $VF_{D,B,i}$ is obtained by the results of the per-bit data-level vulnerability study that essentially provided statistical experimental vulnerability factors for each bit of each combination of data usage types and data sizes.

Trade-off results: Fig. 4.5-4.9 show the reduction of the strongly-protected cache surface against the reduction of the fault coverage of data, when we decide not to protect strongly an amount of LSBs (or MSBs) of the data. Each figure is for a specific combination of data usage type and data size.

These point out how much we can exploit the different vulnerabilities of different bit ranges within application data to reduce the strongly-protected surface of the 64K data cache running the NPB-serial benchmarks to what effect on the fault coverage. The differences between the shown trade-off curves are because the vulnerability characteristics and the rates that the specified application data types occur in the data cache vary among the tested benchmarks.

To obtain the trade-offs shown in Fig. 4.5-4.9, for each combination of application data usage types (D) and data sizes (B), we progressively reduce the protection of D-B data by reducing their bit range under strong protection by 1 bit, until the strong protection has been removed from the full data bit range. Due to the per-bit vulnerability insight that showed less-vulnerable bit ranges either at LSBs or MSBs of application

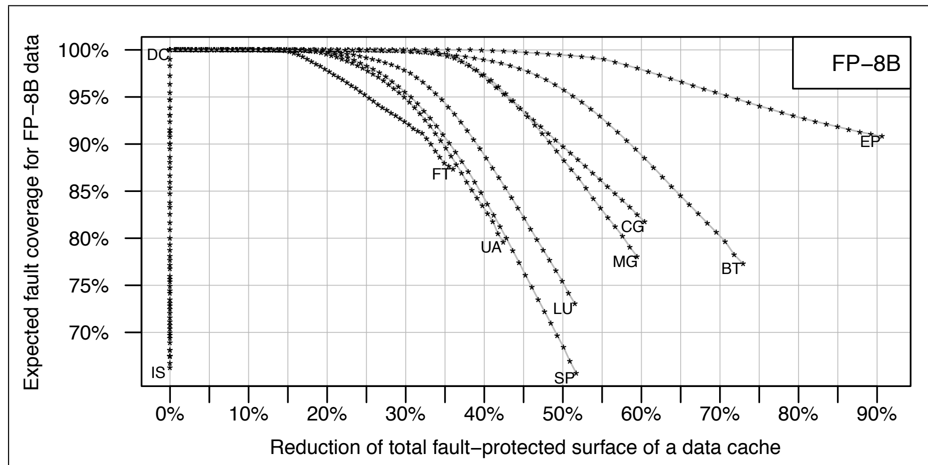

*Figure 4.5: Trading-off fault coverage of **FP (8 byte)** data against the strongly-protected surface reduction (by excepting the FP (8 Byte) **LSBs** from the strongly-protected surface) of a 64K data cache running the NPB-serial benchmarks.*

data, we reduce the bit range under strong protection by 1 bit *in continuous bit ranges starting at the LSB or MSB* (depending on the D-B combination).

Every time the bit range under strong protection of D-B data is reduced, we extrapolate this to the expected total reduction of strongly-protected surface of the data cache. As we know the average occupancy rate of D-B data in the data cache and their individual size (D), it is trivial to estimate the total reduction of strongly-protected surface of the data cache every time we reduce the protection by one bit in every D-B data. Thus, the total possible reduction of strongly-protected cache surface, when unequally protecting D-B data, is bounded by the occupancy rate of D-B data in the data cache. The progressive reduction of the strongly-protected surface can be seen in the figures when moving towards right on the curves.

To simplify the calculation of the strongly-protected cache surface reduction we used the *average* occupancy rates of D-B data. This implies an assumption that each combination of D-B data occupies the data cache in a constant rate over the execution. Although this is not what happens in reality, our assumption stands correct in a valid cache design that could force specific D-B data to occupy specific optimally-sized partitions of the data cache (whose sizes are decided by the occupancy rates) at the cost of possibly increased cache miss rates.

As for the expected fault coverage, each figure shows the expected drop in fault coverage for a specific combination of D-B for the same progressive reduction of their bit ranges under strong protection by one more bit at a time. We computed the fault coverage as discussed before for increasing widths of less-protected bit ranges. Note

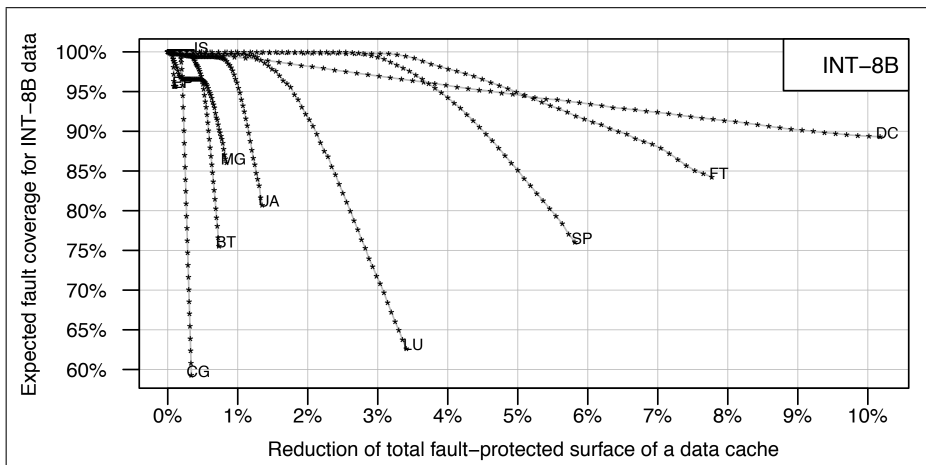

*Figure 4.6: Trading-off fault coverage of **INT (8 byte)** data against the strongly-protected surface reduction (by excepting the INT (8 Byte) **LSBs** from the strongly-protected surface) of a 64K data cache running the NPB-serial benchmarks.*

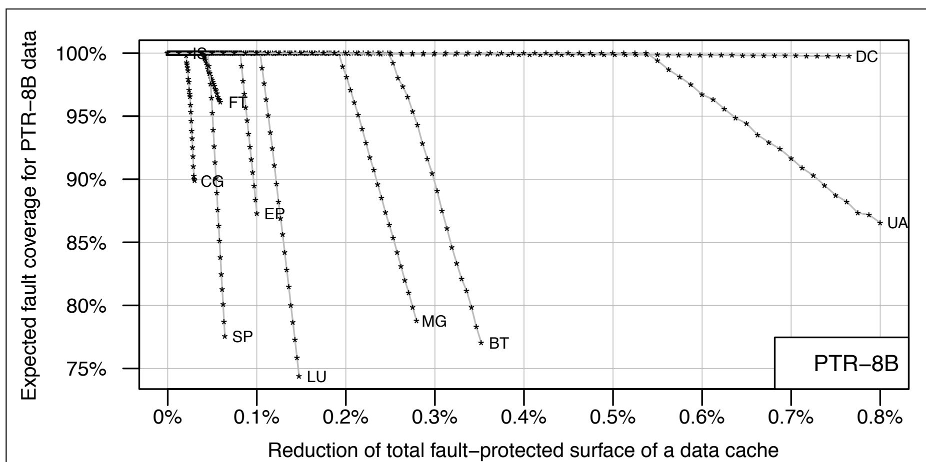

*Figure 4.7: Trading-off fault coverage of **PTR (8 byte)** data against the strongly-protected surface reduction (by excepting the PTR (8 Byte) **MSBs** from the strongly-protected surface) of a 64K data cache running the NPB-serial benchmarks.*

that the estimated fault coverage is estimated for a single D-B data of a given benchmark, and not the full data cache fault coverage, based on the per-bit data-level vulnerability insight. As expected, when reducing the bit range under strong protection of D-B data, the fault coverage of single D-B data in the cache reduces. This can be seen in the figures when moving towards right on the curves.

As an example, in Fig. 4.5, when protecting strongly all bits of FP-8B data of EP (leftmost point), the fault coverage of FP-8B is 100% and the strongly-protected cache surface does not reduce (0% reduction). When progressively reducing the strongly-protected bit range of FP-8B data in EP by removing one LSB at the time, we move

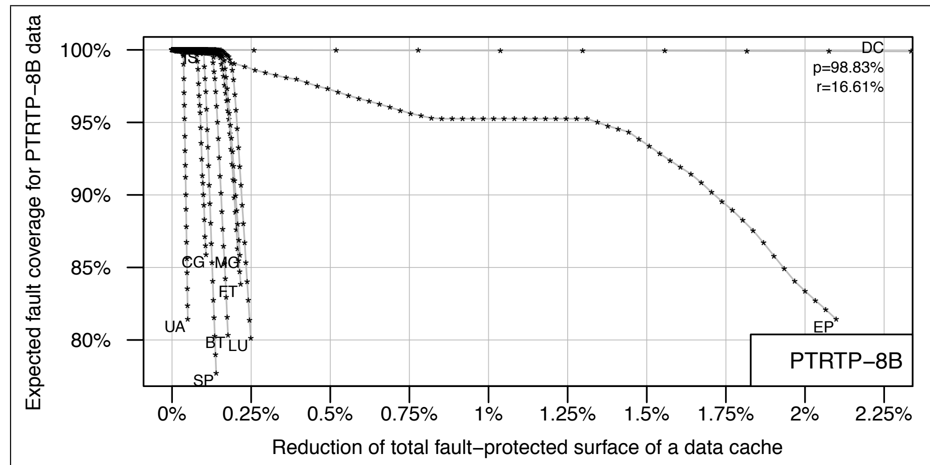

*Figure 4.8: Trading-off fault coverage of **PTRTP (8 byte)** data against the strongly-protected surface reduction (by excepting the PTRTP (8 Byte) **MSBs** from the strongly-protected surface) of a 64K data cache running the NPB-serial benchmarks.*

right on the curve until the full FP-8B are not strongly protected (rightmost point of the EP curve). When the full FP-8B in EP are not strongly protected, as the occupancy rate of FP-8B data in the 64K data cache is 90.64%, the expected strongly-protected surface reduction is also 90.64%. As for the expected fault coverage of FP-8B data in EP, when not protecting their full bit range strongly, it drops to 91.20%. This is obtained by the aforementioned fault coverage calculation method based on the per-bit vulnerability factors observed for the FP-8B data in EP.

Discussion: Out of all the estimated trade-offs, we observe the best trade-offs when unequally protecting floating-point data (FP-8B, Fig. 4.5). As most NPB-serial benchmarks are FP-heavy (except DC and IS), along with the clear per-bit vulnerability patterns of FP data (Fig. 3.7), we can avoid protecting the LSBs of FPs to a greatly reduced total fault-protected surface with a minimal impact on fault coverage. For example, the total fault-protected surface of the data cache under consideration can be reduced by 15% (for 8 out of 10 benchmarks) compared to a fully-protected data cache without affecting the fault coverage of single FP data. If this is to be decided per benchmark, the surface reduction can reach up to 41% (for EP) with a less than 0.01% drop in the fault coverage of its FPs, where 29 LSBs of each FP are not protected in the data cache that is 90.64% occupied on average by FP data during execution of EP.

Furthermore, for a set reliability level we can now reduce the fault-protected surface compared to a fully-protected data cache. For example, the fault-protected surface may be reduced up to 91% (for EP) when not protecting at all the FPs, while the ex-

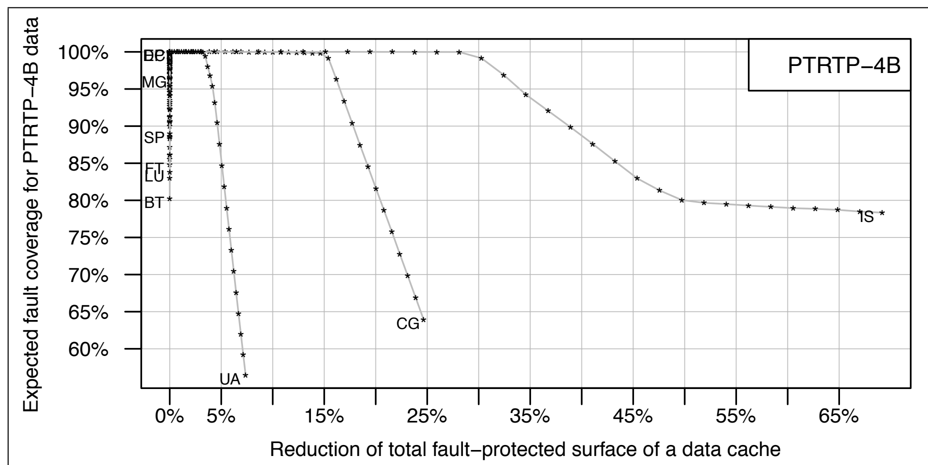

*Figure 4.9: Trading-off fault coverage of **PTRTP (4 byte)** data against the strongly-protected surface reduction (by excepting the PTRTP (4 Byte) **MSBs** from the strongly-protected surface) of a 64K data cache running the NPB-serial benchmarks.*

pected fault coverage for a single FP drops to 90%. The fault coverage, as we defined it per single live data, measures the probability of an SDC caused by a corrupted FP given a fault in a cache location with live FP contents. In reality the probability of an SDC given a fault in a cache location containing a live FP is much smaller but it is not shown as it would obscure the effects of unequal protection on single data.

The same trade-offs are observed for other application data types (INT-8B, PTR-8B, PTRTP-8B, PTRTP-4B) and are shown in Fig. 4.6-4.9 respectively. Despite the fact that these types have clear per-bit vulnerability patterns too, due to their smaller volume in the data cache, avoiding protecting their less-vulnerable bit ranges does not invoke such significant fault-protected surface reduction (except a few limited cases) as it was the case for the FP data. Generally, the vulnerability-driven fault-protected surface reduction is bounded by the average amount of the specific application data types in the data cache for a given benchmark. Nevertheless, unequal protection can be introduced to these types too to add their individual effects for a greater fault-protected surface reduction.

So far, we showed how to exploit the vulnerability characteristics of only specific data types that had clear vulnerability patterns. The same could be done for the remaining data types to further increase the benefits of bit-level unequal protection to the extra cost of identifying all other data types and exploiting them, e.g., by ranking bits of application data based on their per-bit vulnerabilities and reshuffling them accordingly to form less-vulnerable bit ranges.

Table 4.2: Average rates of 'unused' cache contents/space (as percentages of the total cache area) per profiled benchmark in a 64K data cache executing NPB-serial. Rates obtained by the data cache content profiler (Section 4.2).

AVERAGE UNUSED STORES/SPACE RATES (%)
IN A 64K DATA CACHE (4-SET ASSOCIATIVE, 32B BLOCK SIZE)
EXECUTING NPB-SERIAL (3.3.1, INPUT CLASS S)

	BT	CG	DC	EP	FT	IS	LU	MG	SP	UA
Unused stores	16.60	7.21	41.70	4.94	53.07	21.70	39.04	35.11	32.48	41.18
Unused space	8.91	3.59	25.68	1.88	2.42	5.19	5.16	3.56	9.59	5.87
Sum	25.51	10.80	67.38	6.82	55.49	26.89	44.20	38.67	42.07	47.05

Before concluding, it is interesting to note that a significant amount of cache area was reported during the cache content profiling as holding *'unused stored data'* (as shown in Table 4.2 and in Fig. 4.4). As a reminder of how these data were classified as such, during the cache content profiling, a *very short* instruction window of 1000 instructions was imposed on the data classification logic to ensure the completion of the cache content profiler (see Section 4.2). Especially for the case of the memory store accesses, that cannot be classified until accessed by a consecutive memory load access, this short instruction window caused a significant fraction of cache contents to be reported as 'unused stored data' because there was no consecutive memory load access that used them again shortly. This fraction essentially indicates 'unused stored data within 1000 instructions from the moment they were stored in memory' and not necessarily 'stored data that will never be accessed again'. As expected, an increased instruction window would reduce the reported percentage of 'unused stored data' and increase our confidence on the liveliness of the stored data.

Nevertheless, we expect that there is indeed a fraction of cache contents that won't be accessed again in the lifetime of an application. This fraction will be significantly smaller than those reported in Table 4.2 and estimating it would require removing the instruction window limitation. Such cache areas, that store dead data, point out another interesting exploitation area that was left as future work. Given that these dead data cannot affect the execution output, we can avoid strongly protecting them too and reduce further the strongly-protected cache surface, as long as we can guarantee that these data won't be accessed again.

Similar observations hold for the reported *'unused cache space'* by the cache content profiler. This space refers to cache area that was never used or that was storing data whose type has been invalidated (e.g., due to overwriting a part of longer data

by a shorter one, due to a cache line replacement where not all the new cache line's data got accessed and classified, etc.). Therefore this cache area can too be exploited for a further reduction of the strongly-protected cache surface by the reported average 'unused cache space' (Table 4.2).

4.3.2 Practicality Issues of This Case Study

The intention of the presented study was to estimate the maximum potential benefits of exploiting the vulnerability variation of application data. For that purpose, we decided on demonstrating these benefits for the case of a data cache only to answer the question of how much we can exploit the vulnerability characteristics of application data to reduce the strongly-protected surface of a data cache and to what effect on the reliability QoS level.

As this was a limit study, all potential benefits estimated in this chapter assumed an optimal fault-tolerant cache design. The assumed design, on top of being able to drive and protect the data according to their vulnerability, it would have to be able to reconfigure itself on a per application basis to exactly match to the vulnerability characteristics of the executing application. Assuming an available optimal cache design that can support fully-reconfigurable unequal protection is what enabled to get interesting results that show that there is promise in a vulnerability-aware reduction of the strongly-protected cache surface.

Alas, exactly this assumption makes the advocated approach impractical. The practicality of our approach is limited in its attempt to introduce vulnerability-driven unequal protection in the case of an optimal data cache to exploit the per-bit data-level vulnerability characteristics of application data. This is because a one-to-one translation of the potential benefits to actual savings is impossible in the case of a cache design that can offer a reconfigurable strongly-protected cache surface.

Such a design is not trivial to design without negating the savings of unequal protection. A fully reconfigurable design could be possible where, depending on the executing application, the strongly-protected cache surface and protection strength change depending on the vulnerability characteristics of the application. Sadly a major issue of such an approach is that the extra configuration options would negate the savings gained by introducing unequal protection. This is while not considering extra implementation challenges like detecting online the data vulnerabilities, mapping the data to cache locations according to their vulnerability and offering different protection

strength for different data bit ranges. On the other hand, such an approach could provide with the best trade-offs and a general design usable by general purpose CPUs.

An alternative approach would be to remove the reconfiguration attributes of the design and statically configure the cache protection given the executing application vulnerabilities. In such an approach the potential benefits of unequal protection can still be maximized. Though, as this essentially requires a different design per application, it is inapplicable for a general purpose CPU but it could merit such an effort if the cache were to reside in a specialized CPU or a co-processor.

Finally, the most reasonable approach would be a single cache design with predetermined protected cache surface targeting the common case without reconfigurability properties. I.e., a statically configured cache that offers a predetermined set of partitions of different predetermined protection strengths that protect strongly predetermined bit ranges of data. Although this does not offer a lot of flexibility and wouldn't maximize the savings, it would be an improvement on reliability costs, compared to a fully protected cache, and it could work for different workloads. Such an approach though would come at the expense of potentially increased miss rates as it would require a vulnerability-aware mapping of data to specific limited cache locations.

All the above point out the reduced practicality of *exploiting the per-bit data-level vulnerability variations in a fully-reconfigurable vulnerability-aware unequally-protected data cache*, despite showing high exploitation potential as measured by the trade-offs between the strongly-protected surface against fault coverage. This does not negate the importance of the vulnerability characterization results as they can still be applied to other levels/structures that can benefit too by unequal protection.

A suggested alternative approach that shows promise and is left as future work is applying vulnerability-aware unequal-protection to software-level reliability mechanisms. A significant drawback in the case of hardware-level reliability mechanisms has been their lack of flexibility and the requirement of purpose-designed hardware to support new features. On the other hand, software-level approaches could adapt more easily to varying vulnerability characteristics while getting performance benefits. E.g. in an instruction replication based approach, the replication degree may be determined based on the vulnerability and type of the operands in order to get instant performance speedup compared to the fully replicated version.

To summarize, although the vulnerability characterization results showed high exploitation potential, both generally and in the case of a data cache, the practicality of materializing these maximum benefits for the case of a data cache is limited. This holds

more true especially when considering an optimal unequally-protected data cache. A further exploration on how we could translate these benefits into savings for the case of a data cache is left as future work where a practical implementation can be devised that does not compromise the maximum potential savings. Nevertheless, as mentioned, it may be more practical to exploit the vulnerability characterization results in other approaches instead, especially in software-based approaches due to their inherent flexibility.

To conclude, despite the practicality issues of this case study, the observed trade-offs showed that there is potential in shifting from fully equally-protected architectures to vulnerability-aware unequally-protected architectures that exploit the per-bit data-level vulnerability variations to reduce the total fault-protected surface smoothly against the reliability QoS levels.

As the total fault-protected surface can be safely reduced to a minimum effect on the fault coverage, it points out that implementing vulnerability-aware unequal-protection on existing fault-tolerant approaches on a given structure or same-level structures is promising to reduce their reliability overhead and improve their performance.

4.4 Summary

In this chapter we showed the exploitation potential of the characterization insight obtained in Chapter 3.

There are numerous alternative ways, design targets, areas and mechanisms that offer grounds for exploiting the application behavior under data corruption. Out of all the possible alternatives, we focused on exploiting the characterization insight to reduce the reliability-associated overhead of reliability mechanisms with a minimum effect on fault coverage by introducing vulnerability-aware unequal-protection in the case of a data cache by exploiting only the data-related vulnerability variation and, specifically, the per-bit variations.

We chose this exploitation alternative because it was identified as a promising area with high exploitability potential; the per-bit data-level vulnerability variations showed clear and distinct vulnerability patterns, that were consistent among benchmarks (especially for the case of FP data), with highly different vulnerability intensity levels. Moreover, applying this insight for the case of a data cache was possible, as a data

cache can be modified to support vulnerability-driven protection and cache data protection mechanisms can be adapted to support unequal data word protection.

The high exploitability potential of our chosen alternative was further reaffirmed by going through a data cache content profiling process that obtained the occupancy rates of data types within a data cache during the execution of the NPB-serial benchmarks. The cache profiling results showed that there are data types that dominate the data cache in large volumes.

These obtained occupancy rates, along with the per-bit data-level characterization insight, enabled to present a case study to demonstrate how the application data varying vulnerability characteristics can be exploited for reduced-cost unequal fault protection in the case of a 64K data cache running the NPB-serial benchmarks. In particular we explored the potential benefits, when shifting into vulnerability-aware unequal-protection architectures that can trade off between different reliability QoS levels. The potential reliability-overhead benefits were measured in terms of strongly-protected surface reduction against the effect on fault coverage. As our investigation focused on showing the maximum potential, it was agnostic to an actual architectural design and assumed the availability of a fault-tolerant data cache with a protection method that can adapt to unequal protection and a classification mechanism to set the protection levels of application data accordingly.

Despite the practicality issues of implementing this case study, the performed investigation pointed out that there is potential in shifting into vulnerability-aware unequal-protection architectures (by protecting the less-vulnerable areas of a given structure less than its more-vulnerable counterparts) as a viable alternative to exhaustive equal protection in order to reduce the reliability overheads with a minimum effect on a system's dependability.

Chapter 5

Related Work

As we move into deep submicron fabrication technologies new reliability challenges start to appear in an ever-increasing rate in all parts of a system (logic, interconnects, memory elements). Examples include interconnects becoming more susceptible to wire-induced noise, operating voltage/frequency scaling causing timing violations, gamma ray and alpha particle strikes flipping values in transistors, chips and systems deteriorating due to ageing or manufacturing defects.

All such occurrences of anomalous physical conditions are called *hardware faults* [45] and, according to their duration, are classified into transient, permanent and intermittent faults. *Transient faults* (soft errors or single event upsets) are faults that occur only once and do not persist. *Permanent faults* (hard faults) are faults that occur at some point in time and persist from that time onward. *Intermittent faults* are faults that occur repeatedly but not continuously in the same location.

Generally, a set of four steps is followed to tolerate a fault [45]: (1) *Error detection*, where the system becomes aware of the presence of a fault, (2) *Error recovery* (error correction), where the system tries to mask the fault's effect and set the correct expected behavior, (3) *Fault diagnosis*, where the type and location of the fault is identified (not for transient faults), and (4) *Self repair*, where the system is repaired or reconfigured to avoid the diagnosed fault. As each step requires a form of redundancy, fault-tolerant techniques aim to minimize the redundancy-associated performance/cost penalty, while increasing the fault coverage.

The purpose of *fault-tolerant design* is to improve dependability by enabling a system to perform its intended behavior in the presence of a given number of faults [32]. Hardware reliability and correctness is a design constraint equally important to performance and low cost. It is desirable for systems to make forward progress, even in

the presence of faults. As such, a lot of related research exists on ensuring the reliable operation of systems. Furthermore, it is becoming even more imperative to research on reliability as it is a given fact that the transient error rates will always increase as the technologies scale [40].

Fault-tolerant techniques may be classified into hardware level, software level or a combination of both. The level that a technique operates depends on the level where error detection and error correction take place.

Traditional approaches rely on costly physical redundancy to tolerate hardware faults. The most elementary approach to hardware-level fault tolerance is to employ *physical redundancy* [45]. Its simplest form is dual modular redundancy with a comparator. Operations are executed twice and the results are compared to detect faults. This can be further extended into triple modular redundancy, by adding an additional replicated module and replacing the comparator with a voter, to enable also error correction and fault diagnosis. A more general physical redundancy approach is N-modular redundancy for odd values of N greater than three, that as expected provides even better detection, correction and diagnosis capabilities.

Physical redundancy may be implemented at different granularity levels, starting from coarse grain levels, where the whole core is replicated, and moving to finer grain levels, where ALUs or register files are replicated. All such physical redundancy approaches come to a high hardware cost and to high power/energy consumption and, as such, are only used in high-budget and/or mission-critical systems rendering them unacceptable for commodity systems.

Such exhaustive protection is unnecessary for non-life-critical systems as not all hardware faults manifest as the same symptom. Depending on the fault location, the executing workload and the underlying hardware, faults can either (a) get masked by various levels of fault-masking effects (logic-, architectural-, application-level) and result in a correct execution with no visible effects or (b) not get masked and result either in an observable execution upset (application crash, stall or delay) or an unobservable output corruption (silent data corruption - SDC).

Out of all these possible corruption outcomes SDCs are the most severe as they are undetectable online by any means and they do not cause any observable indication that something happened out of the ordinary. The other corruption outcomes are observable and detectable by conventional methods. E.g., software-visible symptom-based fault detection [50, 27, 37, 13, 8] monitors for suspicious behaviors and symptoms,

such as violating likely program invariants, memory exceptions, cache misses, branch mispredictions, fatal exceptions, program crashes, high OS activity, hangs.

Given that not all hardware faults manifest as the same symptom and that it suffices to protect only against SDCs, this thesis targeted a vulnerability-based characterization of application behavior under hardware-induced data corruption to identify areas of lower likelihood to result into an SDC, if a corruption occurs there. As discussed throughout this thesis, such a characterization can help move away from exhaustive equal protection to vulnerability-driven unequal protection of a given structure or same-level structures to get performance/cost benefits without a severe impact on the fault coverage.

In this chapter we present prior works that relate to the goals of this thesis. First, we focus on methods that monitor the behavior of a system under the presence of hardware faults. Then, we discuss about works that implied and/or observed the varying vulnerability characteristics at various levels (with an extra focus on those that investigate variations at data-level and in caches). Afterwards, we present how related works approach the necessary requirements in order to implement a shifted fault-tolerance scheme in a hardware/software co-approach for reliability trade-off purposes. Finally, we discuss on fault-tolerant cache-related approaches, along with those that can adapt to different reliability levels for trade-off purposes.

5.1 Monitoring Behavior Under Data Corruption

Various analytical and experimental techniques have long been proposed for monitoring the effects of hardware faults. These usually target to assess the dependability characteristics of systems and reliability mechanisms and not to drive the design of reduced-cost reliability mechanisms. Nevertheless they demonstrate that not all hardware faults result into the same outcome and provide guidelines on how to capture the behavior under data corruption.

5.1.1 Analytical Techniques

Analytical techniques model the behavior of hardware structures under the presence of faults. Although modeling approaches do not usually correlate the fault's characteristics to the effects on the application behavior, they do observe the varying probability

of a hardware fault in different structures to corrupt the application output. Observing the vulnerability characteristics of hardware structures is more prevalent in works that focus on *vulnerability factor estimation* [31, 46, 47].

Vulnerability factor estimation introduces the concept that not every bit is equally vulnerable and that the vulnerability factor of a structure is the fraction of vulnerable bits within the total bits of the structure. Vulnerability factors can capture both microarchitectural and architectural fault-masking effects (Architectural Vulnerability Factor [31]) in a given structure for a given application through a detailed analysis. In an AVF analysis the bits of the structure that are required for an Architecturally Correct Execution (ACE bits) are constantly tracked while simulating the application. All bits within a structure are assumed as ACE bits, until identified as un-ACE bits, when they fall into categories of identified classes of bits that do not affect the outcome, e.g., bits within predictor structures, the non-opcode bits of a NOP instruction, bits in dynamically dead instructions, etc. In order to compute the actual AVF, this ACE analysis was performed while simulating a set of benchmarks.

Alternative vulnerability factors can estimate the application's tolerance to faults in a specified structure independent of the structure's microarchitectural design (Program Vulnerability Factor [46]) or capture only the microarchitectural fault-masking effects in a hardware structure independent of the application (Hardware Vulnerability Factor [47]).

5.1.2 Experimental Techniques - Fault Injection

As the models in analytical techniques can get large and complex, such analysis is usually slow, offers limited insight into an application execution status and tends to focus on analyzing the behavior of only higher-level hardware structures. An alternative is to use **experimental techniques**, such as experimental verification and error logging, where a system is monitored during its lifetime to record the causes that led to its failures, or full-system simulation under simulated faults.

As these are also time consuming, the experimental approach of **fault injection** has been widely used as a fast straightforward method to test real systems under realistic faults. Fault injection has been mainly used for high-level dependability assessment of reliability mechanisms, systems and applications and it can be hardware implemented [3, 42, 48] or software implemented [20, 42, 48, 7, 39, 17, 14]. Fault-injection experiments perform a set of iterative tests on a system while a given application is

executing. In every iteration, a fault is modeled and the full execution is monitored to determine the exact effect of the injected fault on the application's execution.

In **hardware-implemented fault injection** faults are injected into the hardware layer of the target system either physically by electromagnetic interference and radiation [3] or through the pins of an integrated circuit [42, 48]. Although this injection approach injects real hardware faults that can reach all possible locations, it lacks flexibility and is difficult to operate and control. Furthermore, hardware-implemented fault injection suffers from low portability as it targets specific systems, requires special purpose dedicated hardware to access the tested hardware and can possibly damage the tested system.

On the contrary, **software-implemented fault injection** (SWIFI) overcomes most of these drawbacks by injecting realistic faults using software methods. SWIFI achieves higher properties of repeatability, controllability (in space and time), reproducibility, non-intrusiveness and efficacy [3]. Furthermore, it is less expensive in terms of time and effort and more flexible to repeatedly test and monitor the end-to-end behavior of a system while executing a variety of applications.

SWIFI though cannot emulate all fault models and cannot reach all possible hardware locations that could get corrupted; it is limited into emulating faults in software-visible structures only. Nevertheless software-injected faults can model other high-level models that correspond to other realistic lower-level faults.

Generally, in SWIFI, transient faults are injected by adding traps or replacing instructions, either at pre-runtime or at runtime, depending on the fault trigger and the injection mechanisms. Pre-runtime injection methods mutate the application beforehand, e.g., by substituting instructions and program data by false words [14] or by source code mutation [17]. Runtime injection methods most commonly corrupt memory or register contents using time-based triggers (faults injected at specified times), path-based triggers (faults injected at specified events) or stress-based triggers (faults injected at specified workload threshold levels). The software injection mechanism can be implemented by: direct program memory image corruption [39], dynamic process control structure corruption using debugging registers [20, 48, 7], forcing execution of pre-loaded routines using either software traps [42, 17] or hardware breakpoints of specific target systems [42, 48].

As fault-injection testing captures all levels of fault-masking effects, it has been widely used mainly to evaluate the fault coverage of proposed reliability approaches against the non-protected case [35, 8, 6, 27, 37, 13, 41, 22, 29]. Although usually the

purpose is to measure the fault coverage, these works typically do offer a breakdown of possible execution outcomes per tested workload. Therefore, it is observed that the effect of a hardware fault is not always the same and that it also depends on the inherent fault-tolerance attributes of the workload. On the other hand, these approaches perform limited testing and are usually not thorough enough to correlate the corruption effect to the corrupted location attributes, as they are agnostic to the characteristics of the injected fault.

5.2 Observing Vulnerability Variations

The aforementioned works focused mainly on dependability assessment of reliability mechanisms and not on vulnerability characterization, that was the focus of this thesis. In this section we present prior works that are driven by the varying vulnerabilities of hardware/software structures and investigate possible error sensitivity classifications.

Some fault-tolerant approaches have implied a classification of hardware parts to get cost/performance benefits albeit without a formal exploration of their vulnerabilities.

One such approach, known as *timing speculation*, trades-off error rate against performance/cost in the sense that it pushes the boundaries of the operational frequency or voltage into ranges where timing faults are known to occur. Then, extra hardware mechanisms are added to detect/correct these resulting errors. Timing speculation allows to operate constantly below critical voltage or above critical frequency and gain in cost/performance at the expense of the uncommon case that will fail and will need extra time/support for correction. For this approach to be successful, the added detection/correction hardware must be fast/cheap enough during error-free operation, so that it doesn't negate the overall performance/cost benefits of the timing speculation.

Another timing speculation approach to increase performance relies on pairing two cores in a chip multiprocessor and run the same (or similar) code on both of them [4, 49, 15]. One of the cores will execute speculatively (in other words, in a relaxed functional correctness manner) in order to advance quicker within the application and get execution hints in advance (such as data values, branch outcomes, etc.) that will be communicated to the other core. The other core will execute the same code correctly, without any speculation, but faster due to the received execution hints. This results into both cores running faster combined than either could alone. Meanwhile the non-speculative core, due to its reliable design and always-on nature, will ensure

the correctness of the speculative execution. When throughput is required instead of response time, the pair may be decoupled and operate independently at the safe frequency. Also, as this approach relies on a second core that executes the same code, it offers protection against transient faults too and not only against timing errors.

In the above approaches, that targeted a trade-off between performance/cost and error rate, the faults are mainly caused due to timing violations that are a direct result of the imposed operating frequency/voltage. In order to offer a transparent and manageable fault-tolerant scheme, effort was put into minimizing the number of occurring timing violations by optimizing the common case at the expense of the uncommon case. This is a first indication that not all hardware parts need to be protected equally and that hardware may be classified based on its vulnerability. In particular, timing speculation approaches implied a classification based on the frequency [16, 19] or delay [52, 11] of logic paths by allowing timing/voltage faults on less frequently exercised paths or longer delay paths to gain in cost/performance for the common case.

Similarly, existing cache-specific approaches have implied a vulnerability classification of cache lines based on their access patterns by assigning higher protection priority to the most frequently/recently used cache lines [55] or only to the most recently used cache line in every cache set [22] or only to the dirty cache lines [23]. Such classification is usually obtained through analytical methods or experimental testing. E.g., in [22] fault-injection experiments were performed on a partially protected cache to measure the propagation of injected faults into visible corruptions and to deduce that the most recently used cache lines are more vulnerable.

An analytical alternative to model the behavior of caches under the presence of transient faults is to create a statistical model of a simplified cache and an error propagation model [44]. Faults in both caches and registers are modeled and their propagation is studied taking into account possible application masking effects, e.g., an erroneous cache entry being overwritten by a new value. To test the validity of the proposed error propagation model, a series of fault-injection tests was performed. Although this is not an exact characterization study of data corruption effects on application behavior, it is a promising approach that can be used to estimate the theoretical maximum percentage of corrupted outputs for an application that executes in an unreliable cache.

Further modeling of the architectural masking effects of faults occurring in cache lines has been taken into account in [30] where the mean time to failure of a data cache is computed based on its size and its architectural vulnerability factor, assuming that

the cache fails when two transient faults hit the same line.

More specific fault-injection experiments that try to characterize the effect of data corruption depending on the fault location appear in [34], where soft errors are simulated to perform fault-injection experiments in an instruction cache and a data cache to measure the effects on the execution depending on the location of the injected bit-flip. The injected locations are distinguished (a) between the instruction cache and the data cache and (b) between the cache tag and the cache data. In this approach the injection results showed that the data cache lines and tags are particularly vulnerable, that the vulnerability of the instruction cache lines depends on the application, and that the instruction cache tags are not as vulnerable.

As fault-masking effects occur up to application level, different execution behavior under the presence of hardware faults is expected for different applications and among dependability-aware redesigned versions of the same application [18, 43]. Thus, moving to software abstraction level is promising to observe the varying effects of faults on application behavior [48, 17].

Similarly, fault injection experiments have been used as a way to measure the dependability of applications and show how the workload characteristics affect the behavior under corruption. Such an observation is made in [12], where through fault-injection experiments in a data cache, it is noticed that the resulting error rate depends on the application's input size and the way the application uses the data cache. As the application's input size increases, the error rate increases, until the input size is large enough to fill and fit exactly in the data cache. Then the maximum error rate is observed and, from that point on, as the input size keeps increasing the observed error rate starts decreasing. This behavior is explained by the cache occupancy rates and the cache miss rates, that force the correct values to be fetched from the main memory.

Similar varying vulnerability characteristics are expected among application segments and instructions, as implied in software-level approaches where code segments [9, 35] or individual instructions [6] are marked as critical to set the desired protection level. The same variation can be more formally observed by reliability-aware software analysis to detect statistically-vulnerable code segments [13] and instructions [6, 29].

Moving to application data granularity, it has been implied that the corruption effects vary depending on the data characteristics. For example, data segments can be marked

as approximate if their preciseness is not required for correct execution [38].

In existing cache-specific fault-tolerant approaches that don't protect all cache lines equally, there is an implicit classification of the vulnerability of data based on the probability of being accessed. The most recently accessed and most frequently used variables are considered more error-prone and a higher protection priority must be set for them [22]. The motivation behind this is that most frequently used data are likely to be used again and, if corrupted, are more prone to affect the execution. On the other hand, inactive data have a higher probability of getting replaced/overwritten, thus possible corruption there will get masked. Different methods may be employed to identify such frequently used data. In [55] thresholds are set to define when data should be considered as frequently used or as inactive.

As before, software analysis can be employed to further elaborate on the criticality of application data, e.g., by profiling data according to their liveliness [29] suggesting that more-frequently used data are more likely to corrupt the output. An even finer-grained analysis is performed by the MASK technique [8] that statically analyzes the code to find specific bits within data, that have a known value, that if flipped will certainly cause an application to fail.

Once more, fault injection can be used for the same purpose. In [26], after fault-injection experiments, data segments are marked as non-failure-critical if they affect the output of multimedia workloads, where the output correctness requirements are relaxed. Finally, finer-grained control of the injected fault's location and monitoring of the execution can be used instead to provide more insight on the corruption effects [5]. As an instance, fault injection experiments on specific microarchitectural structures while monitoring the execution, apart from offering a breakdown of outcome types per tested structure, have been used to correlate the fault injection location [27, 37] or the execution outcome [41] to the time between fault injection and fault detection.

5.3 Shifting to a Hardware/Software Co-Approach

As mentioned, it is a given fact that the hardware/software parts are not equally vulnerable to faults. These inherent vulnerability variations may be leveraged to trade-off between reliability and performance/cost in a hardware/software co-approach. Such a trade-off suggests that the reliability levels may be set and defined by the application or the system designer. To be able to set the reliability levels, an interface and communication mechanisms are needed, along with fault-tolerant techniques (either hardware

or software-level techniques) that are configurable to different protection strengths. In this section we present the relevant details of existing works regarding (a) how to set and communicate the desired fault-tolerance levels, (b) how to configure the fault-tolerance levels and (c) how to expose hardware faults to be corrected by the software.

Setting and Communicating the Fault-Tolerance Levels As there is the need to set the desired reliability levels, we will now focus on (a) how the existing works allow the application, developer, compiler or runtime system to set the reliability levels and on (b) how information is communicated between the software and the hardware to set the required QoS reliability levels.

- *API to set fault requirements:* There are a few alternatives on how to specify the API in order to set the desired fault requirements: In [9] code segments are wrapped in code blocks in order to be executed in less-reliable hardware. In [6] the application developer specifies the desired level of fault-tolerance for each instruction or group of instructions, either in high-level language or in assembly code. In [35] the programmer may protect individual code segments and also set the method for error correction. It is a given that all such approaches are accompanied by a respective compiler.
- *Setting fault requirements automatically during compilation:* The aforementioned developer-guided schemes that use a specified API to set the criticality levels are not very practical as the programmer's effort is required. A better alternative is to get the compiler to perform this task automatically, either by performing static analysis or a profile-guided compilation. In [6], it is proposed, among others, a naive automatic compiler scheme that assigns the highest fault-tolerance degree to all control flow instructions and all instructions on which these control flow instructions depend. In [13] the compiler analysis sets fault-tolerance levels by duplicating instructions in only what it considers as vulnerable code segments. In [26], based on fault-injection experiments that provided with vulnerability estimates, the compiler detects/sets the vulnerability of variables thus directing their fault requirements.
- *Communicating fault requirements to the fault-tolerant hardware:* The set fault requirements need to be communicated to the fault-tolerant hardware and this can be done in various ways: By modifying the instruction's encoding to incorporate the required fault-tolerance degree [6]. By extending the ISA with

special instructions to configure the adaptive fault-tolerant hardware to the specified fault-tolerance level [6] or to mark critical and non-critical parts of an application [9]. By introducing new instructions for each combination of original instruction and reliability level [6]. By directly mapping the application data into protected or unprotected hardware based on their vulnerability [26].

Fault-Tolerance Level Configuration: In order to configure the reliability level of the fault-tolerance technique (either for software or hardware techniques), the technique itself must be adaptable to varying degrees of protection strength. E.g., ECC code is able to change in order to detect/correct more or less errors. Such a configuration requirement is not necessary when the protection levels are just non-protection vs. full-protection, as long as the protection mechanism can be turned on and off.

Although there are approaches that dynamically self-tune to the current error rate, they do not count as a configurable fault-tolerant scheme in the way we define it. A method that changes the causes of the faults to control the error rates is different from a method that offers different levels of protection on an existing error rate. Such a self-tuned approach, that adapts to the increased error rates due to voltage scaling, is described in [11] where the error rate is monitored constantly and the supply voltage is dynamically scaled to sustain a reference error rate. Similarly, in [4] the DIVA approach is modified into a self-tuned system that reacts to the current error rate by scaling the operating frequency and voltage.

More interesting are the methods that, for a given error rate, may adapt their fault-tolerance levels. As mentioned, the techniques themselves must be configurable. Most existing approaches that use configurable hardware fault-tolerance techniques set these levels *statically* at design time. E.g., by setting reliable and non-reliable cores [9] or reliable and non-reliable cache lines [26, 23, 22, 55, 23]. On the contrary, software-level techniques lend themselves more naturally to configuration than hardware-level techniques. E.g., in [6] the software fault-tolerance technique of duplicating instructions is used that can be easily configured at design time to specified lower or higher degrees of fault-tolerance by disabling duplication or using triplication respectively.

Exposing Hardware Faults to the Software: All software-only fault-tolerance techniques are effectively exposed to hardware faults. Here we are going to focus only on approaches that explicitly allow a hardware fault to be corrected using software-level fault-tolerance techniques:

- In [9] faults are detected on hardware level and reported to be corrected by the software, only if the software has marked the specific application part to not be corrected at hardware level.

- In [27] a cooperative hardware/software solution is designed that relies on simple low-overhead hardware detectors along with higher-level anomalous software behavior detectors. In order to provide with a scheme that fully protects the system with low overheads, a shift is proposed from detecting a hard fault immediately to detecting the effects of the fault at application level (with low cost software monitors for specific symptoms that were explicitly formulated). This was mainly motivated by the the fact that only hardware faults that propagate and become observable to software need to be handled.

5.4 Fault-Tolerant Cache Design

There has been extensive research on fault-tolerant caches and memories. This is because DRAM and SRAM have always been more susceptible than logic to transient errors. Moreover, due to their relative size comparative to the rest of the processor, faults are more probable to occur in caches and memories. In this section we are going to focus on existing works that relate to our objective of a fault-tolerant data cache architecture that, on top of that, can offer different concurrent levels of protection to trade-off performance/cost against reliability.

Fault Tolerance in Caches: As cache is a structure that holds information, it naturally lends itself to be protected by applying *information redundancy* and using *error-detecting codes* (EDC). The main idea behind information redundancy is to add redundant bits to a dataword to detect when it has been affected by an error [45]. The most common EDC is parity. Odd (or even) parity adds one parity bit to a dataword to convert it into a codeword that now has an odd (or even) total number of ones. With the addition of just one parity bit, it is possible now to detect one single-bit error in the dataword. A natural evolution of parity is to increase the width of the code to detect more that one single-bit errors in the dataword. This is further extended into becoming an *error-correcting code* (ECC) that adds enough redundant bits to also provide correction.

Using EDC and ECC is a simple solution for detecting and correcting errors in stor-

age and offers an easy ground for tradeoffs between reliability and performance/cost just by varying the number of the extra protection bits. Unfortunately, as the EDC/ECC strength is increased, the area and energy overhead grows quickly, while reading and computing the codewords starts to become a performance bottleneck during read operations [21].

To avoid the performance penalties, several alternative approaches have been proposed that vary the way EDC and ECC are applied to the memory hierarchy. All of them put effort on decoupling EDC from ECC, so that the high cost of the infrequent error correction does not affect the common error-free case.

- An approach used by commercial processors [36] is to take advantage of the inherent information redundancy in memory hierarchies by enabling single error-detection code in write-through L1 data caches, along with single error-correcting code in the respective L2 cache.

- Another approach [23] is to store the ECC in a different structure and apply it only to correct dirty cache lines, while the rest clean lines are protected with simple parity EDC. Additionally the dirty cache lines may be periodically cleaned by forcing write-backs to higher level caches.

- In [36] only the EDC bits needed for error detection are added to the L1 cache, in order to keep it small and fast in the error-free case. To support the necessary error correction, the remaining ECC bits are stored separately in a dedicated ECC recovery cache, allowing for higher coverage without affecting the performance of the error-free operation.

  Other similar approaches choose to store the remaining ECC bits in different structures. In [54] the extra correction information is stored in the memory hierarchy as cacheable data, dynamically and transparently partitioning the last level cache into data parts and error codes. In [53] the extra correction information is stored in a dedicated FIFO structure located in low-cost off-chip DRAM without affecting the application's cache behavior.

- An approach that reunites EDC and ECC is proposed in [22], where both EDCs and ECCs are stored into the same dedicated parity cache that holds the correcting codes only for the most recently used cache line per set.

- In [21] a two-dimensional coding scheme is used that decouples error detection from error correction. Conventional per-word horizontal error coding is applied

only for error detection and small-scale error correction, while vertical across-words error coding is used for scalable multi-bit error correction purposes that doesn't affect the latency of normal operation.

ECCs tend to be small enough and correct up to one error to avoid the performance/cost overheads of stronger codes. Furthermore, detection and correction is usually performed when a cache line is accessed. As the time between accesses varies, there is a chance of multiple faults occurring that cannot be detected/corrected by the chosen single-error ECCs. Instead of increasing the strength of the code, [30] introduces the use of *scrubbing* in caches where the cache structure periodically reads each of its entries, corrects any errors, recomputes the ECC and writes the bits back. By introducing this periodical check mechanism, the maximum time between accessing cache data is bounded and the need for increasing the code's strength is avoided.

Fault-tolerant cache approaches may avoid information redundancy and encoding by using the expensive physical redundancy approach and replicate the whole cache structure. An improvement on this scheme is using a shadow cache [22] where replicas of only the most recently used cache line per set are stored to be compared with the ones in the original cache for error-detection purposes.

Another class of approaches that employ a modification of physical redundancy are those that use existing available space to replicate values. In such approaches, cache lines may be replicated in place of dead cache lines and then compare the replicated lines for error detection, while taking care not to evict lines that may be needed or not to incur very high extra power consumption [55].

Other approaches may use the existing available space to substitute cache parts that suffer from hard errors (that are detected offline during booting or periodical tests):

- Faulty cache lines may be disabled completely and substituted into adapted existing structures [2], such as the victim cache, or substituted into original cache space reserved specifically for accommodating faulty cache lines [51].

- Another approach is to disable the faulty cache parts and continue execution in a reduced capacity cache. E.g., complete faulty cache parts may be disabled [25], even if the parts range from a cache line up to the whole cache. Alternatively, faulty cache subblocks may be disabled while the rest cache line is still in use [1]. In any case, when disabling faulty cache parts, it is important to maintain the coverage of the whole address space by the remaining cache parts and take care

to remap the address space in an efficient way so that the performance hit can be highly predictable [1].

- An alternative, where the faulty cache parts are not discarded, is to pair a faulty cache block with another faulty cache block in the same set to form an operational block [24, 51], taking advantage of their non-overlapping faulty parts. Furthermore, extra spare lines are available for completely faulty lines.

Trading-off Performance/Cost Against Reliability in Caches: All presented cache fault-tolerant approaches can trade-off between performance/cost against fault coverage by varying their design parameters, even if this was not their original intent. E.g., as the ECC correction strength can be changed at design time, the discussed ECC-based cache schemes may also become more (or less) robust at the expense of increased (or decreased) performance/cost penalty. Here we are going to focus on cache fault-tolerant approaches that, by design, target to trade-off performance/cost against error coverage and may not protect all cache parts equally to offer multiple concurrent levels of fault tolerance, that preferably can be reconfigured online.

A first attempt to trade-off performance/cost against fault coverage is by protecting only a portion of the cache and essentially providing with two concurrent fault-tolerance levels (protected and non-protected):

- In [26] a partially protected cache is proposed. It is essentially two caches at the same memory hierarchy level where one of them has error-protection mechanisms against transient faults and the other is unprotected. Through fault-injection experiments the vulnerability of classes of data is determined, followed by a compiler analysis to map the data based on their vulnerabilities to the protected or the unprotected cache partition. To further optimize the proposed scheme's coverage and balance the trade-off between cost and fault coverage, a design space exploration strategy is devised to discover the best cache partitioning configuration.

- A similar trading-off approach where not all cache lines are protected is described in [22]. In this approach, the coverage is compromised to reduce area and energy costs of ECC by only protecting the cache lines that have been more recently used in every cache set. Among other considerations, it is suggested that cache lines that are frequently accessed may have a higher probability of corruption due to low noise margins during read/write operations.

Other approaches trade-off performance/cost against fault coverage by protecting some cache lines more than others and essentially providing with two concurrent fault-tolerance levels (protected and stronger-protected):

- In [55] it is proposed to protect all the cache lines with parity code, while replicating the most frequently/recently used ones in place of dead cache lines, in order to offer different concurrent levels of cache fault tolerance. Furthermore, adapting the thresholds, that define when a cache line should be replicated and when a cache line is considered dead, effectively reconfigures the amount of lines that are more protected and also allows for performance/power vs. reliability trade-offs.

- To trade-off area at the expense of performance/bandwidth, in [23] only the dirty cache lines are protected with ECC, while clean cache lines are protected only with EDC. Specifically only one dirty cache line is allowed to be protected with ECC per set in a 4-way set associative cache. If a larger number of lines in a set requires ECC, a write-back is forced to make space for the new ECC.

All approaches that trade-off performance/cost against reliability lend themselves naturally for graceful degradation purposes. The same applies for approaches that treat some values as more important than others. Because caches are mechanisms to improve program performance, they naturally allow to overuse their inherent redundancy and even disable their faulty portions, in order to maintain correctness at the cost of a gracefully degrading performance.

This cache characteristic essentially allows for a smooth trade-off between power/cost and performance, while sustaining the fault coverage at the expense of performance. A scheme that allows for a voltage scaled operation with a performance hit while maintaining high fault-coverage is discussed in [1]. Similarly in [51], two low-overhead schemes are developed to allow operation in lower voltage at the expense of lower cache capacity. One scheme uses pairs of cache lines to form logical cache lines. The other uses a quarter of the cache to store the fault locations and repair bits for the rest of the cache. As a result, two performance levels are offered; either a high-voltage operation with full cache capacity, either a lower-voltage operation with reduced cache-capacity. In both cases, hard faults are detected during boot memory testing and are isolated. Finally, in [25] the impact on the program performance is measured under increasing hard error rates in a data cache. Some degrading strategies are proposed in

order to maintain the cache correctness by disabling faulty cache lines and sets, up to disabling the whole cache. In any case, it is important that the whole address space is covered by the cache, no matter the amount of disabled cache parts.

5.5 Summary

Reliability challenges have long been present in all parts of a system due to occurrences of anomalous physical conditions known as *hardware faults*. To mitigate the resulting reliability concerns, wide research work is present to enable *fault-tolerance properties* in systems.

Traditionally, the reliability approaches relied on costly redundancy to exhaustively protect all parts of a system. As not all hardware fault manifest as the same symptom, this thesis targeted a vulnerability-based characterization of application behavior under data corruption to identify areas of lower vulnerability. The purpose of this characterization was to help avoid the exhaustive protection by shifting to vulnerability-driven unequal protection of a given structure to get performance/cost benefits without a severe impact on fault coverage. For that reason, in this chapter we focused on prior works that related to the goals of this thesis.

First, we presented methods that monitor the effects of hardware faults. These methods are generally classified into analytical and experimental techniques. Analytical techniques model the behavior of hardware structures under the presence of faults, are usually slow, offer limited insight and tend to focus on analyzing the behavior of higher-level hardware structures only. An alternative is to use experimental techniques that rely on monitoring/simulating a system under real/simulated faults. Within the family of experimental techniques, fault injection (either hardware or software-implemented) has been widely used as a faster method to test real systems under realistic faults.

As the analytical/experimental techniques are most commonly used for dependability assessment of reliability mechanisms and not for vulnerability characterization, we proceeded on discussing prior works that are driven by the varying vulnerabilities of hardware/software structures. Some of them imply vulnerability classifications without a formal exploration in order to get performance/cost benefits. Others follow a more formal approach to observe the vulnerability variations, usually through analytical modeling or fault-injection testing. In any case, the variations are observed at different abstraction levels; starting from application level and full processor level,

then going down to caches, code segments and individual instructions, before reaching data-level and bit-level variations.

In order to exploit the varying vulnerabilities, a shift into a vulnerability-driven fault-tolerance scheme is required where one can trade-off the reliability QoS against the performance/cost in a hardware/software co-approach. To do so, we discussed how existing works implement the necessary features to support such a trading-off approach. In particular, (a) how the desired fault-tolerance levels are set and communicated, (b) how the fault-tolerance strength is reconfigured and (c) how to expose hardware faults to be corrected by the software.

Finally, since this thesis focused on measuring the benefits of vulnerability-driven unequal protection in the case of a data cache, we concluded this chapter with the prevalent cache protection approaches and existing fault-tolerant cache designs that can adapt to trade-off reliability QoS against performance/cost.

Chapter 6

Conclusion

Hardware reliability challenges have always been present in all parts of a system manifesting due to occurrences of anomalous physical conditions called *hardware faults*. Hardware faults render the systems unreliable, thus a multitude of related works exists on ensuring the fault tolerance of systems. The purpose of such *fault-tolerant mechanisms* is to improve dependability by enabling a system to perform its intended behavior even in the presence of a given number of faults.

Such reliability solutions always come with associated performance and cost overheads in their effort to ensure the highest possible fault coverage. Traditionally, reliability solutions are aimed to protect equally and exhaustively all hardware parts of a system. This is in order to provide the illusion of a correctly operating hardware. But as shrinking semiconductor technologies come at the cost of higher susceptibility to hardware faults, this approach can no longer be sustainable.

To our benefit, not all hardware faults end up manifesting as errors in the systems behavior and the executing applications behavior. Depending on the fault characteristics (location, type, timing, duration), the executing workload and the underlying hardware, faults can either (a) get masked by various levels of fault-masking effects (logic-, architectural-, application-level) and result in a correct execution with no visible effects or (b) not get masked and result either in an observable execution upset (application crash, stall or delay) or an unobservable output corruption (silent data corruption).

As we face the problem of increasing error rates that induce unsustainable higher reliability costs, especially due to the equal and exhaustive protection, this thesis proposed a solution mainly motivated by the various levels of fault-masking effects. We proposed that we can avoid protecting equally and exhaustively the underlying hard-

ware by breaking the illusion of a fully protected hardware layer. To do so we suggested a shift to vulnerability-driven unequal-protection mechanisms, where the protection strength of a given structure (or same-level structures) is assigned according to the error sensitivity of the components under protection. This means that in order to solve the original problem of high unsustainable reliability costs we shifted the problem into: (a) observing and characterizing the exact end-to-end effects of hardware faults on application behavior, and (b) exploiting the gained insight in a vulnerability-driven reliability mechanism to reduce the reliability overheads.

This chapter concludes this thesis by summarizing in Section 6.1 the proposed approach to reach the thesis goals. Section 6.2 presents our main contributions, while Section 6.3 concludes this thesis with some suggestions for future works.

6.1 Summary

In **Chapter 2** we described our instrumentation-based software-implemented fault-injection (SWIFI) framework for enabling our data-aware characterization study of the exact data corruption effects on application behavior. Our SWIFI framework utilized a single-fault injection tool[1] that is to be repeatedly invoked to inject single-bit transient faults at uniformly chosen memory load accesses during separate runs of an application-under-test. Its purpose is to monitor the application behavior under data corruption, while tracking the corrupted application data, to report detailed diagnostics regarding the corruption characteristics and the corruption effects.

In **Chapter 3** we employed our SWIFI framework to extensively test the NPB-serial benchmark suite (7.8 million fault-injection tests in total) and then aggregated all the detailed reported results in order to observe how application behavior varies under data corruption depending on the characteristics of the corruption and the executing workload.

The amount of performed tests enabled us to, first, present the top-level workload-related vulnerability variations in NPB-serial before moving on to application data related vulnerability variations. We showed how the application behavior in NPB-serial benchmarks varied depending on the characteristics of the corrupted application data; in particular, the size, usage type, user and location in memory of the corrupted data. Then we moved on to a finer-granularity characterization where we showed how the

[1] Additional relevant supplementary material is detailed in **Appendix A**.

application data vulnerability varied depending on the exact bit location of the injected corruption. This allowed to safely identify distinct bit ranges within application data types where the probability to result into an SDC, if corrupted, is very low.

Moreover, given the reported information by our fault-injection tool (relating to the corruption characteristics and the corruption effects) and our original fault model (single bit flips at memory locations just before memory load accesses), we extrapolated the original fault model to model as many other locations of corruption as possible. This enabled to observe the vulnerability variations of NPB-serial benchmarks within the memory space, the register file and among instruction-level characteristics (including instruction type vulnerability variations, program space vulnerability variations and program vulnerability phase detection) without the need to remodel and repeat the experiments.

Generally, the characterization results showed exploitation potential in various locations, even if they are to be used in a per-application basis assuming a previously vulnerability-characterized application. This enabled to proceed to estimate the exploitation potential of the obtained characterization insight.

In **Chapter 4**, after discussing the numerous alternative options that offer grounds for exploiting the application behavior under data corruption, we chose a particular exploitation alternative because it was identified as a promising area with high exploitability potential. To be precise, we focused on exploiting the characterization insight to reduce the reliability-associated overhead of reliability mechanisms with a minimum effect on fault coverage by introducing vulnerability-aware unequal protection in the case of a data cache by exploiting only the data-related vulnerability variation and, specifically, the per-bit variations.

Then we proceeded to present a case study to demonstrate how the application data varying vulnerability characteristics can be exploited for reduced-cost unequal fault protection in the case of a 64K data cache running the NPB-serial benchmarks. In particular we explored the potential benefits when shifting into vulnerability-aware unequal-protection architectures that can trade off between different reliability QoS levels. The potential reliability-overhead benefits were measured in terms of strongly-protected surface reduction against the effect on fault coverage. As our investigation focused on showing the maximum potential, it was agnostic to an actual architectural design and assumed the availability of a fault-tolerant data cache with a protection method that can adapt to unequal protection and a classification mechanism to set the protection levels of application data accordingly.

Despite the practicality issues of implementing this case study, the performed investigation pointed out that there is potential in shifting into vulnerability-aware unequal-protection architectures (by protecting the less-vulnerable areas of a given structure less than their more-vulnerable counterparts) as a viable alternative to exhaustive equal protection in order to reduce the reliability overheads with a minimum effect on a system's dependability.

Finally, in **Chapter 5**, some background information on fault tolerance was discussed followed by prior works that related to the goals of this thesis. The presented relevant studies generally related to methods that investigate the varying vulnerabilities of hardware and software components, methods that attempt to unequally protect a system and methods that trade-off reliability against cost/performance.

6.2 Contributions

In this thesis, we made the following main contributions:

- We established a **portable instrumentation-based SWIFI framework** that can perform extensive tests on target binaries for a data-aware characterization study of the exact data corruption effects on application behavior, where all possible levels of fault-masking effects are captured.

- After **extensive experimental fault-injection tests**, we observed different levels of **vulnerability variations**. The most promising among them was the workload-related vulnerability variation of application data of the NPB-serial benchmarks based on their characteristics, along with the vulnerability variation within parts of application data. For given application data characteristics, we identified clear patterns of less-vulnerable bit ranges that if corrupted are less likely to cause SDCs, e.g., up to 32 LSBs of floating-point data in CG have each less than 1% probability to cause an SDC. The other observed vulnerability variations were in areas such as the memory space, the register file, individual instructions and the program space.

- We demonstrated the **potential exploitability of our data-level characterization findings** in a generic fault-tolerant data cache running the NPB-serial benchmarks. Assuming a vulnerability-aware unequal-protection mechanism we showed how much we can exploit the vulnerability characteristics of application

data and to what effect on the reliability QoS level, e.g., the fault-protected surface of a 64K data cache can be reduced by 41% with a less than 0.01% drop in the fault coverage just by avoiding protection of the less-vulnerable bit ranges of floating-point data in EP.

6.3 Future Works

This section concludes this thesis by sketching some ideas for future works and improvements related to (a) the characterization of application behavior under data corruption and (b) the exploitation of the vulnerability insight. Some of these proposals for future work are the following:

- **Characterizing other applications:** Throughout the thesis the potential in characterizing the application behavior under data corruption was demonstrated only for the case of the serial versions of the programs from NPB. Similar investigation through fault-injection testing is suggested for other types of benchmarks to observe their vulnerability variation and establish if the insight gained in this thesis is observed in other programs too. Equally interesting would be to adapt the SWIFI tool implementation to support multithreaded programs. This can enable the characterization of more programs and also to investigate the vulnerabilities of an extra type of data, those that control the parallel execution.

- **Classification of data types according to their higher-level semantic usage:** We propose augmenting the usage type classification of the SWIFI framework to support detection of higher-level semantic usage information of the application data, i.e., classify them as loop counters, boolean values, pixel colors, parallel execution control values, etc. This could provide more useful information on the vulnerability of data depending on the exact way they are used and could lead to better guidelines regarding which data types to protect stronger or which data types to avoid using at application development or, even, how to drive a compiler-level vulnerability-aware software transformation.

- **Measuring the deviation of reported wrong outputs from the correct expected one:** Given that there is an extra fault-masking level at user level, which we did not consider in this thesis, where the output corruption is not being perceived as a wrong output, e.g., a single wrong pixel color in a frame of a video,

we propose measuring the deviation of reported wrong outputs from the correct expected one. In such a setting, after deciding deviation thresholds, we can stop considering all SDCs as equally severe and further reduce the vulnerable areas that need to be protected.

- **Increasing the propagation tracking capabilities of the characterization framework:** This could help obtain more details on the propagation of the corruption throughout the system to understand how exactly data corruption propagates and where/when exactly does it transform to become an SDC.
- **Machine learning analysis of the experimental fault-injection results:** This could automate the process of identifying the areas of high exploitability potential out of the large amount of data available after the extensive fault-injection experiments.
- **Automatic characterization of application behavior under data corruption:** Instead of extensively testing the full program or memory space of an application, an interesting future work could be to identify program/memory segments that are representative enough so that it suffices to test only those and still be able to get a reliable estimation of the vulnerabilities of the equivalent full application.
- **A practical implementation of a vulnerability exploitation hardware-based approach:** As the maximum potential benefits of exploiting the characterization insight were estimated in this thesis assuming an optimal data cache design, it would be relevant to investigate the design choices and effects in a practical implementation. By that means it could also be investigated how the potential exploitation benefits translate into real-world savings (i.e., area cost, performance, power) and to what effect on the fault coverage. Alternatively, a similar investigation could be performed for a software-based approach.
- **Exploiting the characterization insight in a hardware/software co-approach:** As the characterization insight provides data-level vulnerabilities, it is natural to be applied at software-implemented reliability mechanisms. This can be especially promising if there is communication between the hardware and the software layer to help each other drive their own partial reliability mechanism, i.e., the hardware layer can knowingly let faults propagate to the software layer that could be then become fault-aware and protect against them. Another promising

benefit enabled by such a co-approach is that it can offer a gracefully degrading QoS where, even when the fault rates reach very high rates, the system could configure itself to offer an elementary set of the necessary services.

Appendix A

Single-Fault Injection Tool Output Specification

Every time the single-fault injection tool (as described in Section 2.2) is invoked in fault-injection mode (and not for measuring the fault-free execution time under instrumentation of the application-under-test): (a) it performs a bit-flip in a random (or specified) bit at the memory location accessed by a random (or specified) load memory access, (b) it monitors the effect on the application and (c) it reports back with a CSV record.

The reported information are providing extensive details on the injected fault and the corruption effects on the execution. This appendix details the format and meaning of each value of the reported record.

When a fault was not injected successfully to a specified `memoryRefNo`, the following is reported:

```
memoryRefNo,,,,,,,,,,,,,,,,,,,,,,,,
```

When a fault is successfully injected, the following is reported:

```
memoryRefNo,IPatFault,opcodeAtFault,icountAtFault,
memAddress,size,testedBit,location,user,type,
firstUseReg,firstUseOpcode,firstUseIP,timeToFirstUse,
ifPtrTimeToUse,taintSpreadReg,taintSpreadMem,
maxTaintSpreadReg,maxTaintSpreadMem,totalIcount,outcome,
opcodeAtCrash,crashCode,crashSignal,timeToCrash
```

memoryRefNo: Number of the memory load reference that the fault was injected. It can be either specified as a tool parameter or chosen randomly by the tool. Only memory load references to memory are counted. Just before the desired memory load reference is encountered, the tool injects a bit-flip to the memory address that the memory reference reads.

IPatFault: [hex] Instruction pointer of the instruction when the fault was injected.

opcodeAtFault: [string] Opcode of the instruction when the fault was injected. String produced by Pin's `OPCODE_StringShort()`.

icountAtFault: Number of total instructions executed until the fault was injected.

memAddress: [hex] Memory address of corrupted data.

size: Size of corrupted data in bytes.

testedBit: Location where the bit-flip was injected in the corrupted data. It can be either specified as a tool parameter or chosen randomly by the tool. The LSB is 0. The MSB is $size * 8 - 1$.

location: [char] Type of location in memory address space where the corrupted data reside. 'H' for heap, 'G' for global or 'S' for stack. Default value: 'H'. Type of location is detected based on its memory address. Stack is detected by Pin's instrumentation. Global is detected by checking if the memory address lies within the memory space occupied by the applications currently executing image.

user: [char] User of corrupted data. 'S' for system libraries or 'U' for user. Default value: 'U'. Detected by checking if the memory address lies within the memory space of an applications image that includes `/lib/` or `/lib64/` in its name.

type: [string] Type of corrupted data based on their use. At the moment of the fault-injection (that happens just before a memory read operation), the first written register is used to determine the type of the corrupted value. Alternative detection methods hold for the cases of PTR, PTRMR, PTRTP and INT. Default value: INT.

- **FP:** Floating point (if the first written register is an FP register)
- **IP:** Instruction pointer (if the first written register is the IP register)
- **FLAGS:** If the first written register is eflags/rflags/flags.
- **SEG:** If the first written register is a segment register.
- **PTR:** If the first written register is the stack pointer, or if the uncorrupted value is within the range [lowAddress, highAddress] of one of the applications images (and thus it will probably be a pointer).
- **PTRMR:** If the corrupted value (or the uncorrupted value) is used as an effective memory address to access memory at any point of the execution after the fault injection.
- **PTRTP:** If the corrupted value is used to compute a memory address. This is done by tracking the original corrupted register and corrupted memory locations over a specified window of instructions. In every instruction the corruption is propagating into other registers and memory locations (if needed), until a corrupted register is used for computing a memory address. Then the original corrupted value is assumed to be used as a memory addressing value.
- **INT:** Integer. If none of the above, then the corrupted value is assumed to be used as an integer.

firstUseReg: [string] Register name of the first written register at the moment of fault injection. String produced by Pin's `REG_StringShort()`.

firstUseOpcode: [string] Opcode of the first instruction that uses the `firstUseReg` as an input register, after the fault injection. Default value: UNUSED. String produced by Pin's `OPCODE_StringShort()`. UNUSED when `firstUseReg` was not used at all. MASKED when `firstUseReg` was used as an output register without being used as an input before.

firstUseIP: [hex] Instruction pointer of the first instruction that uses the `firstUseReg` as an input register, after the fault injection. Default value: 0x0

timeToFirstUse: Time in instructions from the fault injection until `firstUseReg` is used as an input register for the first time after the fault injection. Default value: 0

ifPtrTimeToUse: Time in instructions from the fault injection until the corrupted value is detected as PTRTP or PTRMR. Default value: 0.

taintSpreadReg: Number of corrupted registers (due to propagation of the original corruption) at (a) the moment the corrupted value is detected as PTRTP or PTRMR, or (b) the monitoring window is over.[1] Default value: 0

taintSpreadMem: Number of corrupted memory bytes (due to propagation of the original corruption) at (a) the moment the corrupted value is detected as PTRTP or PTRMR, or (b) the monitoring window is over.[1] Default value: 0

maxTaintSpreadReg: Total number of registers corrupted (due to propagation of the original corruption) from the moment of fault injection until (a) the moment the corrupted value is detected as PTRTP or PTRMR, or (b) the monitoring window is over.[1] Default value: 0

maxTaintSpreadMem: Total number of corrupted memory bytes (due to propagation of the original corruption) from the moment of fault injection until (a) the moment the corrupted value is detected as PTRTP or PTRMR, or (b) the monitoring window is over.[1] Default value: 0

totalIcount: Total number of instructions executed until the application stopped execution regardless of the outcome.

[1] Note that `taintSpreadReg`, `taintSpreadMem`, `maxTaintSpreadReg`, `maxTaintSpreadMem` are computed only when tracking the corrupted value to check if it is used for memory addressing. Even if the corrupted value is not detected as PTRTP, if these values are not zero, then they can still provide an insight of how much the corruption propagated during the propagation tracking. In other words, when these are non-zero, they provide with information regarding how much the corruption propagated between the fault-injection moment until one of the following happened: (a) PTRMR detected, (b) PTRTP detected, (c) monitoring window is over.

outcome: [string] Execution outcome of experiment.

- **CO:** **CO**rrect output
- **CD:** **C**orrect output but **D**elayed, detected based on the total executed instructions according to the set delayedRatio
- **WR:** **WR**ong output
- **SI:** application **S**tall detected due to excessive executed **I**nstructions, according to the set unresponsiveRatioInstr
- **ST:** application **S**tall detected due to excessive execution **T**ime, according to the set unresponsiveRatioTime
- **CR:** application **CR**ashed

opcodeAtCrash: [string] Opcode of last instruction executed that caused the crash. Only valid when application crashed due to a signal. Default value: “X”. String produced by Pin’s `OPCODE_StringShort()`.

crashCode: Application exit code. Only valid when application exited abnormally. Default value: -1

crashSignal: Crash signal number. Only valid when application crashed due to a signal. Default value: -1

timeToCrash: Time in instructions from the fault injection until the crash. Only valid when the application crashed (due to a signal or abnormal exit). Default value: -1

Example:

```
...,CO,X,-1,-1,-1
...,CD,X,-1,-1,-1
...,WR,X,-1,-1,-1
...,SI,X,-1,-1,-1
...,ST,X,-1,-1,-1
...,CR,X,crashCode,-1,timeToCrash
...,CR,opcodeAtCrash,-1,crashSignal,timeToCrash
```

Bibliography

[1] Jaume Abella, Javier Carretero, Pedro Chaparro, Xavier Vera, and Antonio González. Low Vccmin fault-tolerant cache with highly predictable performance. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-42, pages 111–121, New York, NY, USA, 2009. ACM.

[2] Jaume Abella, Eduardo Quiñones, Francisco J. Cazorla, Yanos Sazeides, and Mateo Valero. RVC: A mechanism for time-analyzable real-time processors with faulty caches. In *Proceedings of the 6th International Conference on High Performance Embedded Architectures and Compilers*, HIPEAC '11, pages 97–106. ACM, January 2011.

[3] Jean Arlat, Yves Crouzet, Johan Karlsson, Peter Folkesson, Emmerich Fuchs, and Günther H. Leber. Comparison of physical and software-implemented fault injection techniques. *IEEE Transactions on Computers*, 52(9):1115–1133, 2003.

[4] Todd M. Austin. DIVA: A reliable substrate for deep submicron microarchitecture design. In *Proceedings of the 32nd ACM/IEEE International Symposium on Microarchitecture*, MICRO-32, pages 196–207, Washington, DC, USA, 1999. IEEE Computer Society.

[5] Fatemeh Ayatolahi, Behrooz Sangchoolie, Roger Johansson, and Johan Karlsson. A study of the impact of single bit-flip and double bit-flip errors on program execution. In *Computer Safety, Reliability, and Security*, volume 8153 of *Lecture Notes in Computer Science*, pages 265–276. Springer Berlin Heidelberg, 2013.

[6] Demid Borodin, Ben H.H. Juurlink, Said Hamdioui, and Stamatis Vassiliadis. Instruction-level fault tolerance configurability. *Journal of Signal Processing Systems*, 57(1):89–105, October 2009.

[7] Joao Carreira, Henrique Madeira, and Joao Gabriel Silva. Xception: a technique for the experimental evaluation of dependability in modern computers. *IEEE Transactions on Software Engineering*, 24(2):125–136, Feb 1998.

[8] Jonathan Chang, George A. Reis, and David I. August. Automatic instruction-level software-only recovery. In *Proceedings of the 2006 International Conference on Dependable Systems and Networks*, DSN '06, pages 83–92, June 2006.

[9] Marc de Kruijf, Shuou Nomura, and Karthikeyan Sankaralingam. Relax: An architectural framework for software recovery of hardware faults. In *Proceedings of the 37th International Symposium on Computer Architecture*, ISCA '10, pages 497–508, New York, NY, USA, 2010. ACM.

[10] NASA Advanced Supercomputing Division. NAS Parallel Benchmarks. `http://www.nas.nasa.gov/publications/npb.html`, 2012.

[11] Dan Ernst, Nam Sung Kim, Shidhartha Das, Sunjay Pant, Rajeev Rao, Toan Pham, Conrad Ziesler, David Blaauw, Todd Austin, Krisztian Flautner, and Trevor Mudge. Razor: A low-power pipeline based on circuit-level timing speculation. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-36, pages 7–18, Dec. 2003.

[12] F. Faure, R. Velazco, M. Violante, M. Rebaudengo, and M.S. Reorda. Impact of data cache memory on the single event upset-induced error rate of microprocessors. *IEEE Transactions on Nuclear Science*, 50(6):2101 – 2106, December 2003.

[13] Shuguang Feng, Shantanu Gupta, Amin Ansari, and Scott Mahlke. Shoestring: Probabilistic soft error reliability on the cheap. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '10, pages 385–396, New York, NY, USA, 2010. ACM.

[14] Jean-Philippe Gerardin. The DEF.Injecto test instrument, assistance in the design of reliable and safe systems. *Computers in Industry*, 11(4), 1989.

[15] Brian Greskamp and Josep Torrellas. Paceline: Improving single-thread performance in nanoscale CMPs through core overclocking. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*,

PACT '07, pages 213–224, Washington, DC, USA, 2007. IEEE Computer Society.

[16] Brian Greskamp, Lu Wan, Ulya R. Karpuzcu, Jefrey J. Cook, Josep Torrellas, Deming Chen, and Craig Zilles. BlueShift: Designing processors for timing speculation from the ground up. In *Proceedings of the IEEE 15th International Symposium on High Performance Computer Architecture*, HPCA '09, pages 213–224, Feb. 2009.

[17] Martin Hiller, Arshad Jhumka, and Neeraj Suri. PROPANE: An environment for examining the propagation of errors in software. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA'02, pages 81–85, 2002.

[18] Kuang-Hua Huang and Jacob A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computers*, C-33(6):518–528, June 1984.

[19] Andrew B. Kahng, Seokhyeong Kang, Rakesh Kumar, and John Sartori. Designing a processor from the ground up to allow voltage/reliability tradeoffs. In *Proceedings of the IEEE 16th International Symposium on High Performance Computer Architecture*, HPCA '10, pages 1–11, Jan. 2010.

[20] Ghani A. Kanawati, Nasser A. Kanawati, and Jacob A. Abraham. FERRARI: A flexible software-based fault and error injection system. *IEEE Transactions on Computers*, 44(2):248–260, February 1995.

[21] Jangwoo Kim, Nikos Hardavellas, Ken Mai, Babak Falsafi, and James Hoe. Multi-bit error tolerant caches using two-dimensional error coding. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-40, pages 197–209, Washington, DC, USA, 2007. IEEE Computer Society.

[22] Seongwoo Kim and Arun K. Somani. Area efficient architectures for information integrity in cache memories. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, ISCA '99, pages 246–255, Washington, DC, USA, 1999. IEEE Computer Society.

[23] Soontae Kim. Area-efficient error protection for caches. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '06, pages 1282–1287, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[24] Cheng-Kok Koh, Weng-Fai Wong, Yiran Chen, and Hai Li. Tolerating process variations in large, set-associative caches: The buddy cache. *ACM Transactions on Architecture and Code Optimization*, 6(2):8:1–8:34, July 2009.

[25] Hyunjin Lee, Sangyeun Cho, and Bruce R. Childers. Performance of graceful degradation for cache faults. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*, ISVLSI '07, pages 409–415, Washington, DC, USA, 2007. IEEE Computer Society.

[26] Kyoungwoo Lee, Aviral Shrivastava, Ilya Issenin, Nikil Dutt, and Nalini Venkatasubramanian. Partially protected caches to reduce failures due to soft errors in multimedia applications. *IEEE Transactions on Very Large Scale Integrated Systems*, 17(9):1343–1347, September 2009.

[27] Man-Lap Li, Pradeep Ramachandran, Swarup K. Sahoo, Sarita V. Adve, Vikram S. Adve, and Yuanyuan Zhou. Understanding the propagation of hard errors to software and implications for resilient system design. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 265–276, New York, NY, USA, 2008. ACM.

[28] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[29] Mojtaba Mehrara and Todd Austin. Exploiting selective placement for low-cost memory protection. *ACM Transactions on Architecture and Code Optimization*, 5(3):14:1–14:24, December 2008.

[30] Shubhendu S. Mukherjee, Joel Emer, Tryggve Fossum, and Steven K. Reinhardt. Cache scrubbing in microprocessors: Myth or necessity? In *Proceedings of*

the 10th IEEE Pacific Rim International Symposium on Dependable Computing, PRDC '04, pages 37–42, Washington, DC, USA, 2004. IEEE Computer Society.

[31] Shubhendu S. Mukherjee, Christopher Weaver, Joel Emer, Steven K. Reinhardt, and Todd Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-36, pages 29–, Washington, DC, USA, 2003. IEEE Computer Society.

[32] Victor P. Nelson. Fault-tolerant computing: Fundamental concepts. *Computer*, 23(7):19–25, July 1990.

[33] University of Edinburgh. Edinburgh Compute and Data Facility (ECDF). `http://www.ecdf.ed.ac.uk`.

[34] Maurizio Rebaudengo, Matteo Sonza Reorda, and Massimo Violante. An accurate analysis of the effects of soft errors in the instruction and data caches of a pipelined microprocessor. In *Proceedings of the Conference on Design, Automation and Test in Europe*, volume 1 of *DATE '03*, Washington, DC, USA, 2003. IEEE Computer Society.

[35] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I. August. SWIFT: Software implemented fault tolerance. In *Proceeding of the International Symposium on Code Generation and Optimization*, CGO '05, pages 243–254, March 2005.

[36] Nathan N. Sadler and Daniel J. Sorin. Choosing an error protection scheme for a microprocessor's L1 data cache. In *Proceeding of the International Conference on Computer Design*, ICCD '06, pages 499–505, October 2006.

[37] Swarup Kumar Sahoo, Man-Lap Li, Pradeep Ramachandran, Sarita V. Adve, Vikram S. Adve, and Yuanyuan Zhou. Using likely program invariants to detect hardware errors. In *Proceedings of the 2008 IEEE International Conference on Dependable Systems and Networks*, DSN '08, pages 70–79, June 2008.

[38] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. EnerJ: Approximate data types for safe and general low-power computation. In *Proceedings of the 32nd ACM SIGPLAN Conference*

on Programming Language Design and Implementation, PLDI '11, pages 164–174, New York, NY, USA, 2011. ACM.

[39] Z. Segall, D. Vrsalovic, D. Siewiorek, D. Yaskin, J. Kownacki, J. Barton, R. Dancey, A. Robinson, and T. Lin. FIAT - Fault injection based automated testing environment. In *Proceedings of the 18th International Symposium on Fault-Tolerant Computing*, FTCS-18, pages 102–107, June 1988.

[40] Premkishore Shivakumar, Michael Kistler, Stephen W. Keckler, Doug Burger, and Lorenzo Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, DSN '02, pages 389–398, 2002.

[41] Alex Shye, Tipp Moseley, Vijay Janapa Reddi, Joseph Blomstedt, and Daniel A. Connors. Using process-level redundancy to exploit multiple cores for transient fault tolerance. In *Proceedings of 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '07, pages 297–306, June 2007.

[42] Daniel Skarin, Raul Barbosa, and Johan Karlsson. GOOFI-2: A tool for experimental dependability assessment. In *Proceedings of the 2010 IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '10, pages 557–562, June 2010.

[43] Joseph Sloan, David Kesler, Rakesh Kumar, and Ali Rahimi. A numerical optimization-based methodology for application robustification: Transforming applications for error tolerance. In *Proceedings of the 2010 IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '10, pages 161–170, July 2010.

[44] Arun K. Somani and Kishor S. Trivedi. A cache error propagation model. In *Proceedings of the Pacific Rim International Symposium on Fault-Tolerant Systems*, PRFTS '97, pages 15–21, Washington, DC, USA, 1997. IEEE Computer Society.

[45] Daniel J. Sorin. Fault tolerant computer architecture. *Synthesis Lectures on Computer Architecture*, 4(1):1–104, 2009.

[46] Vilas Sridharan and David R. Kaeli. Eliminating microarchitectural dependency from architectural vulnerability. In *Proceedings of the IEEE 15th International*

Symposium on High Performance Computer Architecture, HPCA '09, pages 117–128, February 2009.

[47] Vilas Sridharan and David R. Kaeli. Using hardware vulnerability factors to enhance AVF analysis. In *Proceedings of the 37th annual International Symposium on Computer Architecture*, ISCA '10, pages 461–472, New York, NY, USA, 2010. ACM.

[48] David T. Stott, Benjamin Floering, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. NFTAPE: A framework for assessing dependability in distributed systems with lightweight fault injectors. In *Proceedings of the 4th International Computer Performance and Dependability Symposium*, IPDS '00, pages 91–, Washington, DC, USA, 2000. IEEE Computer Society.

[49] Karthik Sundaramoorthy, Zach Purser, and Eric Rotenburg. Slipstream processors: Improving both performance and fault tolerance. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS-IX, pages 257–268, New York, NY, USA, 2000. ACM.

[50] Nicholas J. Wang and Sanjay J. Patel. ReStore: Symptom based soft error detection in microprocessors. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks*, DSN '05, pages 30–39, Washington, DC, USA, 2005. IEEE Computer Society.

[51] Chris Wilkerson, Hongliang Gao, Alaa R. Alameldeen, Zeshan Chishti, Muhammad Khellah, and Shih-Lien Lu. Trading off cache capacity for reliability to enable low voltage operation. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 203–214, Washington, DC, USA, 2008. IEEE Computer Society.

[52] Gil Wolrich, Edward McLellan, Larry Harada, James Montanaro, and Robert A.J. Yodlowski. A high performance floating point coprocessor. *IEEE Journal of Solid-State Circuits*, 19(5):690–696, October 1984.

[53] Doe Hyun Yoon and Mattan Erez. Flexible cache error protection using an ECC FIFO. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 49:1–49:12, New York, NY, USA, 2009. ACM.

[54] Doe Hyun Yoon and Mattan Erez. Memory mapped ECC: Low-cost error protection for last level caches. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 116–127, New York, NY, USA, 2009. ACM.

[55] Wei Zhang, Mahmut T. Kandemir, Anand Sivasubramaniam, and Mary Jane Irwin. Performance, energy, and reliability tradeoffs in replicating hot cache lines. In *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '03, pages 309–317, New York, NY, USA, 2003. ACM.

[56] Yu Zhu, Jaeyeon Jung, Dawn Song, Tadayoshi Kohno, and David Wetherall. Privacy Scope: A precise information flow tracking system for finding application leaks. Technical report, EECS-2009-145, Department of Computer Science, UC Berkeley, 2009.