University of New Mexico
UNM Digital Repository

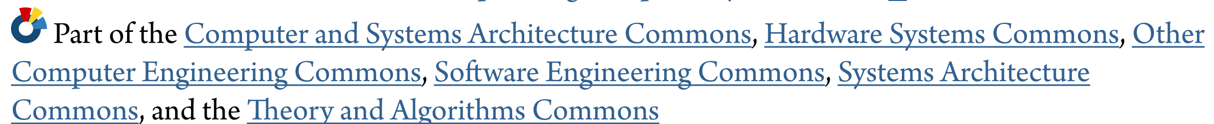
Computer Science ETDs | Engineering ETDs

Fall 11-15-2018

Criticality Assessments for Improving Algorithmic Robustness

Thomas B. Jones
University of New Mexico

Follow this and additional works at: https://digitalrepository.unm.edu/cs_etds



Part of the Computer and Systems Architecture Commons, Hardware Systems Commons, Other Computer Engineering Commons, Software Engineering Commons, Systems Architecture Commons, and the Theory and Algorithms Commons

Recommended Citation
Jones, Thomas B.. "Criticality Assessments for Improving Algorithmic Robustness." (2018). https://digitalrepository.unm.edu/cs_etds/96

This Dissertation is brought to you for free and open access by the Engineering ETDs at UNM Digital Repository. It has been accepted for inclusion in Computer Science ETDs by an authorized administrator of UNM Digital Repository. For more information, please contact disc@unm.edu.

Thomas Berry Jones

Candidate

Computer Science

Department

This dissertation is approved, and it is acceptable in quality and form for publication:

Approved by the Dissertation Committee:

David Ackley, Chairperson

Darko Stefanovic

George Luger

Thomas Caudell

Lance Williams

Criticality Assessments for Improving Algorithmic Robustness

by

Thomas Berry Jones

B.S., Computer Engineering and Mathematics, Gonzaga University, 2010

M.S., Computer Science, University of New Mexico, 2013

DISSERTATION

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Doctorate of Philosophy

Computer Science

The University of New Mexico

Albuquerque, New Mexico

December, 2018

Dedication

To my family, Bryan, Colleen, Tim, and Jon, for their love and support and to my nephew and niece Colby and Olivia.

A Vehicle, Heinsight, and Carbon on Paper

Acknowledgments

I would first like to thank my friends, especially James Orrell, Kyle Wilson, and Evan Poncelet who have, at various times, encouraged me to continue on this path. I don't think I would be here today without them. I would also like to thank my mother, Colleen Jones, who graciously helped with copy-editing the final document on very short notice.

Next, I want to thank my committee members, who have all been very kind and have had great suggestions for improving the work. Dr. George Luger has provided immense intellectual, moral, and material assistance during this process. He has always supported my work and has helped me greatly with my career. I am sure that I would not have finished without his assistance. Dr. Darko Stefanovic was very gracious to join my committee with little notice and has taught me a great deal about programming languages. Additionally, I want to thank Dr. Lance Williams who has always inspired some of the most interesting ideas when I've spoken with him. His suggestions concerning failure redistribution methods were spot on. I would also like to thank Dr. Thomas Caudell who has been very benevolent in looking after the work of this wayward graduate student after his retirement.

Finally, I would like to thank my advisor, Dave Ackley, for putting up with me all these years. I have learned more from him than from anyone else.

Criticality Assessments for Improving Algorithmic Robustness

by

Thomas Berry Jones

B.S., Computer Engineering and Mathematics, Gonzaga University, 2010

M.S., Computer Science, University of New Mexico, 2013

Ph.D., Computer Science, University of New Mexico, 2018

Abstract

Though computational models typically assume all program steps execute flawlessly, that does not imply all steps are equally important if a failure should occur. In the "Constrained Reliability Allocation" problem, sufficient resources are guaranteed for operations that prompt eventual program termination on failure, but those operations that only cause output errors are given a limited budget of some vital resource, insufficient to ensure correct operation for each of them.

In this dissertation, I present a novel representation of failures based on a combination of their timing and location combined with criticality assessments—a method used to predict the behavior of systems operating outside their design criteria. I observe that strictly correct error measures hide interesting failure relationships, failure importance is often determined by failure *timing*, and recursion plays an important role in structuring output error. I employ these observations to improve the output error of two matrix multiplication methods through an economization procedure

that moves failures from worse to better locations, thus providing a possible solution to the constrained reliability allocation problem. I show a 38% to 63% decrease in absolute value error on matrix multiplication algorithms, despite nearly identical failure counts between control and experimental studies. Finally, I show that efficient sorting algorithms are less robust at large scale than less efficient sorting algorithms.

Contents

List of Figures **xiii**

List of Tables **xxii**

Glossary **xxiii**

1 Introduction **1**

1.1 Background 1

1.1.1 From Fault Tolerance to Approximate Computing 3

1.1.2 Best-Effort Computing 4

1.2 Problem Statement 6

1.3 Conceptual Framework 7

1.4 Research Questions and Hypotheses 10

1.4.1 How do Measures of Correctness Hide or Uncover Interesting Failure Dynamics Inside Algorithms? 10

1.4.2 How Are an Algorithm's Failure Dynamics Related to the Algorithm's Known Behavior? 11

1.4.3 Can Failure Dynamics Analysis Improve Algorithmic Performance in Resource Constrained Environments? 11

1.4.4 Which Operations Respond Well to This Method? 12

1.5 Procedures 13

1.6 Thesis Outline 14

2 Related Work **15**

2.1 Criticality 15

2.2 Fault Tolerance 17

2.3 Fault Injection 19

2.4 Quantified Correctness 21

2.5 Approximate Algorithms and Computing 23

2.6 Error Economization 26

2.7 Theories of Robustness 27

3 The Model **29**

3.1 Author Contribution Statement 29

3.2 Publication Notes 29

3.3 Introduction 30

3.4 Criticality Assessments 31

3.5 Error Measures 34

3.6 Mathematical Abstractions of Space and Time . . . 35

3.7 Criticality Defined . . . 37

3.8 Criticality Explorer . . . 40

3.9 Failure Shaping . . . 43

4 Sorting Algorithm Criticality **47**

4.1 Author Contribution Statement . . . 47

4.2 Publication Notes . . . 47

4.3 Introduction . . . 48

4.3.1 Measuring Sortedness . . . 50

4.4 Results . . . 51

5 Criticality and Armoring in Matrix Multiplication **58**

5.1 Author Contribution Statement . . . 58

5.2 Publication Notes . . . 58

5.3 Introduction . . . 59

5.4 Failure Shaping on Scalar Multiplication . . . 60

5.4.1 Criticality Assessment Results on Scalar Multiplication . . . 62

5.4.2 Failure Shaping Results on Scalar Multiplication . . . 62

5.5 Failure Shaping Matrix Multiplication . . . 63

5.5.1 Criticality Assessment Results on Matrix Multiplication . . . 64

5.5.2 Failure Shaping Results on Matrix Multiplication 65
5.6 Proxy Failure Shaping 66
5.6.1 Criticality and Failure Shaping Results on Scalar Multiplication Proxy Method 66

6 Scalable Robustness 97

6.1 Author Contribution Statement 97
6.2 Publication Notes 97
6.3 Introduction 98
6.4 Scalable Robustness Defined 99
6.4.1 Average-Case Scalable Robustness 99
6.4.2 Worst-Case Scalable Robustness 102
6.5 Scalable Robustness on Pairwise-Comparison Sorting 104
6.5.1 Algorithms Old and New 105
6.6 Average-Case Scalable Robustness on Sorting Algorithms 106
6.7 Worst-Case Scalable Robustness on the Sorting Algorithms 111
6.7.1 Quicksort—Spearman's Footrule Error 114
6.7.2 Bubble Sort—Spearman's Footrule Error 114
6.7.3 Round Robin Sort—Spearman's Footrule Error 115

7 Discussion 120

7.1 Summary of Work 120

7.2 Questions Answered 121
7.2.1 Strict Correctness Hides Criticality Structure While Quantified Correctness Reveals It 122
7.2.2 Criticality Structure Seems Related to Known Algorithmic Behaviors 123
7.2.3 Algorithmic Performance in Resource-Constrained Environments Is Improved Through Failure Dynamic Analysis 127
7.2.4 Bulk Program Operations Make Good Choice for Failure Shaping 131
7.2.5 Numerical Stability 131
7.2.6 When Will Criticality Structures Scale Effectively? 132
7.3 Limitations of the Study 132
7.4 Future Work 135
7.4.1 Expanded Empirical Studies 135
7.4.2 Scaling Through Search 135
7.4.3 Machine Learning and Attribution 136
7.4.4 Hardware Failure Studies 138
7.4.5 Spatial Graphs for Coordinated Failures 138
7.4.6 Stochastic and Continuous Failure Redistribution Policies . . . 138
7.4.7 Scalable Robustness Beyond Sorting 139
7.5 Significance 139
7.6 Conclusion 141

A Permission to Reproduce Previously Published Content 143

A.1 IEEE Reproduction Instructions 145

A.2 Springer Nature License Terms and Conditions 148

References 154

List of Figures

3.1 **Criticality Assessment and Failure Shaping Overview.** *Criticality Explorer* takes a prepared algorithm with error measures, input generator, and annotated method level failure interfaces shown on the left. After a calibration step to evaluate the maximum failure point count, and samples per failure point, the tool then estimates failure point criticalities through a Monte Carlo simulation. Each sample produces the output of a single run with, and without, a failure injected on the a given failure point and scores the error between the two runs. The errorful inputs, outputs, and absolute difference score can be found in the bottom center of the figure. Failures are then shaped using the median criticality and error scores are produced for average i.i.d. as well as shaped failures. See Section 3.8 for details. Figure reprinted from [1] with permission. 46

4.1 **Strict Correctness Criticality on Sorting Algorithms** Extremal values dominate in a plot of strict correctness criticality (y axis) vs. the comparisons executed during a sort ('Comparison Index'; x axis): Most faults are either critical or not critical. See Section 4.4 for details. Figure reprinted from [2] with permission. 54

4.2 **Quantified Correctness on Merge Sort** The average conditional positional error curves (*top* graph), corresponding to the estimated error with and without the fault at the given comparison index, and the positional error criticality (*middle* graph), both based on a 10% background error rate. Note that the purple and blue boxes are error bars. See text for details. Figure reprinted from [2] with permission. 55

4.3 **Quantified Correctness Results on Quicksort** In quicksort we see that the choice of error measure can effect which comparisons are seen as most critical. With a positional error measure the first N comparisons are the *most* critical. However, under the inversions error measure the first N comparisons are the *least* critical comparisons. See text for details. Figure reprinted from [2] with permission. 56

4.4 **Periodic Criticality in Bubble Sort** When facing a non-zero background error rate bubble sort presents periodic criticality behavior. The bottom graph is a subgraph of the top graph. See text for details. Reprinted from [2] with permission. 57

5.1 **Scalar Multiplication Criticality Assessment Results.** *Absolute log* criticality assessment results for both the naive and Karatsuba multiplication algorithms. The left graphs show results on the `common add` failure interface, the middle graphs show results for the `check` failure interface, and the right graphs show results for the `Karatsuba add` interface, which only the Karatsuba algorithm called. The top graphs are for input size $N = 100$ while the bottom graphs are for size input $N = 500$. The naive multiplication algorithm shows a log-linear growth pattern in criticality on the `check` operation and a log-linear growth pattern on half of the `add` operation while the Karatsuba algorithm shows a group of three maxima on every interface. Note that graphs have different x and y axes. See text for details and Figures 5.2 (page 69), 5.3 (page 70), 5.4 (page 71), 5.5 (page 72), 5.6 (page 73), and 5.7 (page 74) for expanded views of each graph. Reprinted from [1] with permission. 68

5.2 **Expanded Scalar Multiplication Criticality Assessment Results on NaiveMultiply.add (N=100).** See Figure 5.1 (page 68) for details. 69

5.3 **Expanded Scalar Multiplication Criticality Assessment Results on NaiveMultiply.check (N=100).** See Figure 5.1 (page 68) for details. 70

5.4 **Expanded Scalar Multiplication Criticality Assessment Results on KaratsubaMultiply.add (N=100).** See Figure 5.1 (page 68) for details. 71

5.5 **Expanded Scalar Multiplication Criticality Assessment Results on NaiveMultiply.add (N=500).** See Figure 5.1 (page 68) for details. 72

5.6 **Expanded Scalar Multiplication Criticality Assessment Results on NaiveMultiply.check (N=500).** See Figure 5.1 (page 68) for details. 73

5.7 **Expanded Scalar Multiplication Criticality Assessment Results on KaratsubaMultiply.add (N=500).** See Figure 5.1 (page 68) for details. 74

5.8 **Scalar Multiplication Failure Shaping Results on Absolute Value.** Average error rates for both the naive and Karatsuba multiplication algorithms assessed using *average absolute value* error measure on input size 100. Assessments were performed on both algorithms using a baseline i.i.d. failure model and the median value failure shaping technique. See text for details. Reprinted from [1] with permission. 75

5.9 **Scalar Multiplication Failure Shaping Results on Log Absolute Value.** Average error rates for both the naive and Karatsuba multiplication algorithms assessed using *average log absolute value* error measure on input size 100. Assessments were performed on both algorithms using a baseline i.i.d. failure model and the median value failure shaping technique. See text for details. Reprinted from [1] with permission. 76

5.10 **Scalar Multiplication Failure Shaping Results on Absolute Percent Value.** Average error rates for both the naive and Karatsuba multiplication algorithms assessed using *average absolute percentage value* error measure on input size 100. Assessments were performed on both algorithms using a baseline i.i.d. failure model and the median value failure shaping technique. Percent value is represented as a number in $[0, 1]$ and is sensitive to minor changes in its denominator. See text for details. Reprinted from [1] with permission. 77

5.11 **Matrix Multiplication Criticality Assessment Results.** Criticality results for both naive and Strassen's matrix multiplication. The left graphs show *log Frobenius norm*, the middle graphs show *Frobenius norm*, and the right graphs show *infinity norm* results. The graphs on the top are for matrix input size 4, the middle set are at size 8 and the bottom set are at size 16. Naive results are omitted in the four graphs towards the bottom right to emphasize similar structures in Strassen's algorithm at multiple scales. Note that x and y axes do not match for every graph. Each error measure is valued in its own units. See text for details and Figures 5.12 (page 80), 5.13 (page 81), 5.14 (page 82), 5.15 (page 83), 5.16 (page 84), 5.17 (page 85), 5.18 (page 86), 5.19 (page 87), and 5.20 (page 88) for expanded views of each graph. Reprinted from [1] with permission. 79

5.12 **Expanded Log Frobenius Norm Matrix Multiplication Criticality Assessment Results on NaiveMultiply.check (N=4).** See Figure 5.11 (page 79) for details. 80

5.13 **Expanded Frobenius Norm Matrix Multiplication Criticality Assessment Results on NaiveMultiply.add (N=4).** See Figure 5.11 (page 79) for details. 81

5.14 **Expanded Infinity Norm Matrix Multiplication Criticality Assessment Results on NaiveMultiply.check (N=4).** See Figure 5.11 (page 79) for details. 82

5.15 **Expanded Log Frobenius Norm Matrix Multiplication Criticality Assessment Results on NaiveMultiply.check (N=8).** See Figure 5.11 (page 79) for details. 83

5.16 **Expanded Frobenius Norm Strassen's Criticality Assessment Results on NaiveMultiply.add (N=8).** See Figure 5.11 (page 79) for details. 84

5.17 **Expanded Infinity Norm Strassen's Criticality Assessment Results on NaiveMultiply.check (N=8).** See Figure 5.11 (page 79) for details. 85

5.18 **Expanded Log Frobenius Norm Matrix Multiplication Criticality Assessment Results on NaiveMultiply.check (N=16).** See Figure 5.11 (page 79) for details. 86

5.19 **Expanded Frobenius Norm Strassen's Criticality Assessment Results on NaiveMultiply.add (N=16).** See Figure 5.11 (page 79) for details. 87

5.20 **Expanded Infinity Norm Strassen's Criticality Assessment Results on NaiveMultiply.check (N=16).** See Figure 5.11 (page 79) for details. 88

5.21 **Focused Log Frobenius Norm Criticality Results.** Log Frobenius norm structures continue to persist at larger input scales. See text for details. Reprinted from [1] with permission. 89

5.22 **Matrix Multiplication Proxy Method Criticality Assessment Results on Input Size N=4.** Criticality results for both naive and Strassen's algorithms on the scalar multiplication failure interface using *Frobenius norm* error measure at matrix input size N=4. See text for details. Reprinted from [1] with permission. 90

5.23 **Matrix Multiplication Proxy Method Criticality Assessment Results on Input Size N=8.** Criticality results for both naive and Strassen's algorithms on the `scalar multiplication` failure interface using *Frobenius norm* error measure at matrix input size N=8. See text for details. Reprinted from [1] with permission. . . . 91

5.24 **Matrix Multiplication Proxy Method Criticality Assessment Results on Input Size N=16.** Criticality results for both naive and Strassen's algorithms on the `scalar multiplication` failure interface using *Frobenius norm* error measure at matrix input size N=16. See text for details. Reprinted from [1] with permission. . . . 92

5.25 **Matrix Multiplication Proxy Failure Shaping Results on Frobenius Norm.** Average error rates for both the naive and Strassen's algorithms assessed using *Frobenius norm* error measure at input size 16. Assessments were performed on both algorithms using a baseline i.i.d. failure model, the direct failure shaping technique, and the proxy method failure shaping technique. We see error reductions between 38% and 63%. See text for details. Reprinted from [1] with permission. 93

5.26 **Matrix Multiplication Proxy Failure Shaping Results on Infinity Norm.** Average error rates for both the naive and Strassen's algorithms assessed using *infinity norm* error measure at input size 16. Assessments were performed on both algorithms using a baseline i.i.d. failure model, the direct failure shaping technique, and the proxy method failure shaping technique. See text for details. Reprinted from [1] with permission. 94

5.27 **Matrix Multiplication Proxy Failure Shaping Results on Log Frobenius Norm.** Average error rates for both the naive and Strassen's algorithms assessed using *frobenius norm* error measure at input size 16. Assessments were performed on both algorithms using a baseline i.i.d. failure model, the direct failure shaping technique, and the proxy method failure shaping technique. See text for details. Reprinted from [1] with permission. 95

5.28 **Proxy Criticality Assessment and Failure Shaping Results on Input Size 32.** On the left I present the criticality assessment for proxy failure interface `scalar multiply` on the *infinity norm* error measure. It presents similar structures at input size 32 as at smaller input scales. On the right I present proxy failure shaping results for size 32 without direct failure shaping results. See text for details. Reprinted from [1] with permission. 96

6.1 **Summary of Empirical Results.** Measured average error rates for merge sort (**MS**), quicksort (**QS**), full bubble sort (**FBS**), and round robin sort (**RRS**), assessed according to strict correctness error (**SCE**, left), max displacement error (**MDE**, middle), and Spearman's footrule error (**SFE**, right, note changed y scale), plotted vs the failure rate and at input sizes from 100 to 100,000 (**1e2**..**1e5**). See text for details. Reprinted from [3] with permission. 112

6.2 **Merge Sort WCSR Strategy:** The attacker faults comparisons with the largest item until the list is shaped with all the largest items excluding the largest item (Z) in the first half, and all the smallest items preceded by Z in the second half. The final merge compares items across the red-dashed line. Reprinted from [3] with permission. 113

7.1 **Selected Leverage Results.** *Log absolute value* leverages on both naive and Karatsuba scalar multiplication algorithms are presented on the left. *Frobenius norm* leverage results on the matrix multiply algorithms are presented on the right. Note that Karatsuba leverage on `check` and `add` operations seems to dip before picking up and growing above naive scalar multiply's `check` leverage at scale 500. Strassen's algorithm leverage on all operations grows faster than naive matrix multiply's leverage on any operation. Note that graphs have different x and y axes. See text for details. Reprinted from [1]. 129

List of Tables

6.1 **Summary of $\boldsymbol{ACSR_{iid}}$ Results.** Where known, each (sorting algorithm, error measure) assessment is marked with a bound on its expected error measure growth rate—$O()$ upper bounds for assessments proven $ACSR_{iid}$, and $\Omega()$ lower bounds for those proven *not* $ACSR_{iid}$. Those marked **?** are currently unproven, though empirical data (e.g. Fig. 6.1) suggests they are all $ACSR_{iid}$. Reprinted from [3] with permission. 106

6.2 **Summary of *WCSR* Results.** Each (algorithm, context) pair marked with an X is provably *not WCSR*. The $\Theta(N^2)$ round robin sort algorithm, when paired with the Spearman's footrule error context, stands out as the only algorithm that is *WCSR*. Reprinted from [3] with permission. 119

Glossary

approximate algorithms and computing A computing technique in which programs may return inaccurate results in return for the ability to complete a computation under circumstances where a correct result is difficult or impossible to obtain.

best-effort computing A paradigm in which computational units try to improve the current program state without working towards a complete and precisely correct solution.

C_f Notation for a computation C running with **failure pattern** f.

correct mode An operation's expected computational behavior when there is no failure.

criticality A method for measuring the importance of a component in the context of a whole system.

criticality assessment A technique for assessing algorithm operation **criticality** in some **context**.

context See **program context**

ϵ Notation for an i.i.d. error probability.

error economization The purposeful redirection of resources from one part of a computation to another with the goal of decreasing output errors measured via an **error measure**.

error measure Any function that accepts both a correct and an incorrect program output and returns a scalar difference between them.

execution index A count of the number of times an operation or method has been called during a single program run.

experiment object A Java object that has an experiment method that runs a program and a score method that acts as an *error measure*.

failure dynamics The interlocking set of relationships between correct and failed operations and the output of a program.

failure interface See method level failure interface.

failure method A method that implements a **failure mode** and must have the same signature as the original correct method except that it accepts an additional input for a random number generator and has a signifier, such as 'Rand', appended to its method name.

failure mode A description of an operation's behavior during a failure, used to inject failures into a program.

failure pattern A record of when and where each failure in a program occurs.

failure pattern distribution A probability distribution over the set of all possible **failure patterns**.

failure point A tuple of (**execution index**, operation) that uniquely defines a single failure during a single program run.

failure shape The **criticality** distribution on a **field of failure points** produced by a **criticality assessment**.

fault injection The simulation of hardware and software failures that can then be be used to test program behavior under faulty conditions.

fault tolerance The silent correction of hardware or software faults at run time such that the high level algorithm or user never observes the fault.

fault-intolerance/fault-tolerance A paradigm that combines **fault tolerance** techniques that silently correct failures with fault intolerance that kills a program or process whenever a failure is not corrected.

field of failure points A set of **failure points** over one or more operations and execution indexes—large enough to contain most possible runs of an algorithm at some input scale. Each point in the field has an associated criticality.

$f \sim F$ Notation for a **failure pattern** f drawn from a **failure pattern distribution** F.

$f \wedge K^*$ Notation for a **failure pattern** f, modified so that operation K is operating in its **failure mode**.

input generator A generator that produces random **input objects**.

input object An object with a `randomize` method that acts as an input generator.

K^* Notation for operation K operating in its **failure mode**.

leverage The average **criticality** of all **failure points** with a criticality above the median criticality divided by the average criticality of those beneath the median.

method level failure interface A Java method annotated with an '@Randomize' tag and with an accompanying **failure method**. At runtime faults are injected into the program by calling the failure method instead of the original **correct mode** method.

program context Paired with a specific program, a context is a list of operations that can fail in the program and their **failure modes**, an **input generator** for the program, and an **error measure** on the program.

presortedness A method for measuring how close a list is to sorted, a kind of **error measure**.

quantified correctness Any measure of error on a program, in some context, such that the number of possible output error values grows with a function that is in $\omega(C)$ in program input size.

scalable robustness The property of a program, in some context, such that the error of the program approaches zero as the number of failures in the program approaches zero while the input size of the program approaches infinity.

theories of robustness Computational theories that relate deviations in program inputs and operations, compared to some expected baseline for either, to a measure of error on the program output.

Chapter 1

Introduction

In this thesis I present a method for measuring operation importance on failure-prone hardware, providing a novel perspective on the computational process and paving the way for robustness resource economizations that improve the behavior of programs in fallible environments. This work grows out of a concern for problems introduced by silent data corruptions (SDC) which are observed both in large scale and embedded computing environments [4, 5]. These problems call into question paradigms that rely on the removal of failures for computational success and lead to a lack of understanding concerning the impact of failures when they are inevitable.

1.1 Background

At the beginning of the computer revolution computational devices were unreliable. Based on vacuum tubes and electromechanical switches, they could break down and corrupt a computation after a very short period of time [6]. It has become a cliche in computer science that the word 'bug' entered the lexicon when an insect landed on Harvard University's Mark II calculator and caused the computer to glitch. While the word 'bug' can be found applied to tiny technical errors in a system or engine

as early as the 19th century [7], it is nevertheless true that one moth's unfortunate encounter with an Mark II electronic panel is one of its first uses in the field of computer science [8]. Failures like this were common for early computer scientists, who placed a lot of effort into building machines of sufficient accuracy to provide useful results.

In response, the pioneers of our field developed a paradigm of ruthless correctness to combat hardware failures. In 1948 John von Neumann gave a series of lectures in which he explained how this paradigm grew out of engineers' lack of understanding about errors and opposed it to the behavior of computations in natural systems:

> Natural organisms are sufficiently well conceived to be able to operate even when malfunctions have set in. They can operate in spite of malfunctions, and their subsequent tendency is to remove these malfunctions... [In an artificial automaton any malfunction] represents a considerable risk that some generally degenerating process has already set in within the machine. It is, therefore, necessary to intervene immediately, because a machine which has begun to malfunction has only rarely a tendency to restore itself, and will more probably go from bad to worse... With our artificial automata we are moving much more in the dark than nature appears to be with its organisms... [We are] much more "scared" by the occurrence of an isolated error and by the malfunction which must be behind it. Our behavior is clearly that of over-caution, generated by ignorance [6].

Von Neumann's solution was to combine the natural and artificial paradigms at different computational levels. Computer engineers would assemble modules composed of multiple copies of fallible components. If the results of a supermajority of the components were the same then the answer they provided would be accepted. This allowed for masking some failures as in the 'natural automata' paradigm. If

there was no supermajority agreement, however, then the result was considered invalid, a fault was *detected*, and the whole computation was thrown out or rewound to the last correct state, and a human was informed of the error in keeping with the methods for correcting 'artificial automata'. Assuming *independent and identically distributed* (i.i.d.) failures amongst the components, this method could drive the module's *undetected* failure rate to an arbitrarily low value by growing the number of redundant modules, given that per component failure rates were below a very conservative 16% [9].

1.1.1 From Fault Tolerance to Approximate Computing

While the two-level failure correction method introduced by von Neumann did not have a name in his day, today we call it the *Fault-intolerant/fault-tolerant* paradigm (fault-tolerance) [10, 11]. Computations are designed first to avoid failures whenever possible and terminate on any hardware failure that may introduce a software fault, hence the fault-intolerance, while striving to preserve and protect a fragile core of correct computation by re-performing and correcting failures that may occur at run time using fault tolerance techniques. These methods may be based in hardware, such as [12–15] where, for example, special chip designs allow for multi-threaded result checking, or they may be in software such as [16–19] where extra instructions are added to the normal software flow to ensure correct execution. This paradigm has allowed for the construction of the architectures, built around 'guaranteed' deterministic hardware, that have 'guaranteed' deterministic execution.

However, fault tolerance/intolerance tends to create a bifurcation in the way computer and software engineers think about computational processes. Programs are often written with large nonlinearities where the manipulation of a few bits of data can greatly impact final outcomes. Software engineers can remain ignorant of how these nonlinearities are impacted by failures that occur at a lower level as

those failures are expected to disappear under a contract of hardware-software stack determinism that relies on ensuring that data looks correct. However, this subjects the system to arbitrary impact due to SDCs since these failures, by definition, 'look correct' [20].

While these techniques work for the worst problems at today's computing scales, solutions are often achieved through a slow evolutionary procedure in which failures must first become a problem *to* be solved before they *can* be solved. Many times, this game of catch-up works fine. Occasionally, it does not [21]. Sometimes, the cure is almost worse than the sickness, demanding immense growth in resources to suppress only occasional failures as in process replication or checkpoint-restart in high performance computing (HPC) environments. When implemented naively, these techniques can lead to massive increases in energy usage making it more difficult to meet important environmental and energy cost goals [22–24]. However, as long as we do not understand failure such methods are necessary.

Power wastage is also a pressing problem for embedded computations. To tackle this problem, researchers working with these devices are considering hardware techniques, such as voltage over-scaling, that trade accuracy for lower hardware energy use [25]. Conceivably, in the future, chips will present a reliability-efficiency interface that may be used by economizing software to regulate chip resource on a per program counter tick basis or redirect less important computational steps to a "green" computational core that uses less energy [26]. However, these developments that drop computational determinism must eventually lead to a reconsideration of what it means for a program to be "correct".

1.1.2 Best-Effort Computing

One approach to dealing with computational correctness and computing beyond determinism is *best-effort computing* [27, 28], in which hardware promises only that

its operations will typically be correct, while guaranteeing neither full correctness nor any specific distribution of failures. One encouraging step in this direction are the fields of *approximate algorithms and computing*—computing techniques in which programs may return possibly inaccurate results rather than generate a guaranteed accurate result, in return for the ability to complete a computation under circumstances where a correct result is difficult or impossible to obtain [5, 29–31]. It has already shown promise in applications such as approximate video processing (e.g. [32, 33]), where even large failures may produce only minor output errors, such as periodic blips on the screen.

Approximate computing, however, sometimes requires that technology choices be evaluated only in a whole-systems context, risking chicken-and-egg problems if actual end-use data is expected to inform the system's design. For example, past energy conservation approximation techniques have relied on detailed hardware knowledge to limit their search space when finding the best economization for a given system application [33]. The application of approximate computing to general computations requires techniques that can provide economizations outside of a particular hardware context.

A priori theoretical analysis of algorithms could lead to some useful conclusions about algorithmic behavior in the face of undetected failures. This is true of efficiency analysis, which can sometimes be re-purposed for analyzing algorithm robustness. In some cases, these analyses are not that difficult. For example, we can prove that the first N comparisons of quicksort are the most important because of how the first pivot moves numbers across half the list length on average, while pivots later in the algorithm tend not to move numbers as far. However, many theoreticians are not focused on questions of *failure dynamics*—the interlocking set of relationships between correct and failed operations and the output of an algorithm, but rather seek to ensure absolutely correct outputs within some error bounds, in keeping with the fault tolerance paradigm.

The pursuit of correctness certainly seems respectable, but it implies that all errors are equally bad, no matter how harmless or catastrophic. A better way is to switch from a paradigm of strict correctness to one of *quantified correctness* which allows for measuring continuous or nearly continuous degrees of output error. Standardized error measures can be developed for common object classes, in the same way that other methods are integrated into objects today. For example, when sorting lists a *presortedness* measure (e.g. [34, 35]) can be re-appropriated to measure the degree of successful sorting in the final outcome. These measures can then be used on many list-like objects. Similarly, absolute difference measures can be employed to judge the error on algorithms that produce integer, floating point, and matrix outputs.

Strict correctness presents us with a weak feedback signal for computational error. Abandoning strict correctness may seem like a high price to pay; indeed, the fault-tolerance paradigm will eventually lead to the evolutionary embedding of knowledge concerning failure within the code itself. However, we cannot say how efficient that process will be, or if it will ever completely illuminate the darkness surrounding computational error that von Neumann spoke of so long ago.

1.2 Problem Statement

Since traditional software engineering assumes deterministic execution, there is a gap in knowledge concerning:

- *which* operations in a program are most critical to obtain high-quality results,
- *what* can be done to ensure successful program operation in resource constrained environments, and
- *how* the properties of a program, such as its efficiency or the ordering of its

operations, interact with its robustness in the face of failures.

To answer these questions I present the "constrained reliability allocation" problem. In this problem sufficient resources are guaranteed for operations that may prompt eventual program termination or infinite loops on failure while operations that only cause output errors are given a limited budget of some vital resource. It is then the job of some economization procedure to distribute these resources so that program output is minimally damaged.

Therefore, the contributions of this dissertation are threefold:

1. it quantifies the importance of operation failures in a traditionally deterministic group of algorithms on steps that appear to lie somewhere between 'negligible impact' and 'program critical',
2. it provides a framework and method for improving the performance of deterministic algorithms in environments where there are not sufficient resources for guaranteed deterministic operation, and
3. it develops a deeper understanding of the relationship between algorithmic efficiency and robustness using a newly developed scalable robustness paradigm.

Now we turn to the notions of computational robustness and error measurement that form this dissertation's foundation.

1.3 Conceptual Framework

This study grows out of ideas related to criticality, fault tolerance, fault injection, quantified correctness, approximate computing, error economization, and theories of efficiency and robustness. I seek to use criticality, measured through quantified-correctness evaluations of error on fault-injected programs, to show how approximate

computations on faulty hardware can be be economized. Using the insights thus gained, I then propose a theory of computational robustness that stands in contrast to those posed by the fault tolerance paradigm.

I will touch, briefly, on each of these concepts here, but a more in-depth exploration of each subconcept can be found in Section 2.

Criticality is a method for measuring the importance or value of a component in the context of a system [36–39]. A simple example of criticality might be the minimum throughput on some node in a flow graph experiencing a max flow event or the change in device state caused by receiving or not receiving, a particular message. It can be viewed as a statistical derivative, measuring the expected change in the output of a system given a change in some internal component.

Fault Tolerance/Intolerance, as explained, is the bimodal paradigm whereby failures are either suppressed through some statistical method, or alternatively, if they are detected but uncorrected they cause a system crash or restart.

Fault Injection is a technique whereby computational operations are modified at run time to behave differently than expected. Faults can be injected either at the hardware level, by manually adding components capable of changing electrical states, or at the software level through various techniques with virtual machine fault injection being a popular technique [40].

Quantified Correctness means evaluating program or system performance in a manner that goes beyond the 'is correct/is not correct' level. It also includes stepping beyond measuring the simple probability of correct program responses. Rather, under quantified correctness a single program run is capable of having error responses that range over a large number of different values with a cardinality that is $\omega(C)$ in functions of program input size.

Approximate Algorithms and Computing, as previously explained, are computing

frameworks where program outputs are not expected to be correct, but are rather allowed to vary from the correct value within some error range. At first *approximate algorithms* were applied to solving problems with no known feasible solution, such as NP-complete computations [41], but recently *approximate computing* has been applied to situations where hardware level errors are allowed to percolate through the computational stack all the way to program output [42].

Error Economization is the purposeful redirection of resources from one part of a computation to another with the goal of decreasing output errors measured via an **error measure**.

Theories of Efficiency account for the amount of resources required to successfully implement a computing strategy for solving some problem. They are very common in the computing world. The big-O, big-Θ, and big-Ω notations and their attendant concepts come to mind. There are also *Theories of Robustness* that account for the output error of a program given some error on program inputs or distribution of failures on program operations. However, these are less generalized and famous than those concerning computational efficiency. An example of such a theory is program continuity found in [43] where the authors propose that a program in continuous if a small change in the input leads to a small change in the output. Alternatively, in [44] the authors present a model of robustness which focuses on δ, the number of allowed failures per unit time.

Using this framework it becomes possible for us to ask some serious questions and hypothesize about the failure dynamics of programs subject to the constrained reliability allocation problem.

1.4 Research Questions and Hypotheses

A full and complete consideration of failure dynamics is outside the scope of this study. When programs begin to fail there are many strange phenomena that can occur, including some failures that improve program performance or combinations of failures that do far greater or less damage than can otherwise be accounted for by their individual impact [45]. However, there are some general questions about these dynamics and their relationship to computational robustness and efficiency that I do ask:

1.4.1 How do Measures of Correctness Hide or Uncover Interesting Failure Dynamics Inside Algorithms?

Any study about failure dynamics should be concerned with the method of measuring error on the output of an algorithm. While I have indicated that quantified correctness is a possible path forward for understanding how computations fail, it does not seem to be obvious to the field that quantified correctness measures should be employed when judging program performance. This opposition, traditional all-or-nothing correctness against quantified correctness suggests two subquestions.

How do all-or-nothing error measures impact the observed failure dynamics of an algorithm?

To answer this question, this dissertation evaluates algorithmic failure dynamics with error measures that use a strictly correct evaluation procedure to judge performance as a baseline on sorting algorithms.

How do error measures that allow for partial credit on continuous or multi-step scales, such as quantified correctness measures, impact our understanding of algorithmic failure dynamics?

After measuring a subset of failure dynamics with all-or-nothing error measures this dissertation empirically examines both sorting algorithms and multiplication algorithms with quantified correctness measures.

My primary hypothesis is that strict correctness measures often hide an algorithm's failure dynamics while quantified correctness measures expose interesting failure dynamics. This is due to the tendency of strict error measures to treat all failures equally and mask the failure's impact on algorithmic output. This hypothesis will be falsified if strictly correct error measures produce similar failure dynamics results as quantified correctness measures or if the failure dynamics exposed by quantified correctness measures are otherwise uninteresting.

1.4.2 How Are an Algorithm's Failure Dynamics Related to the Algorithm's Known Behavior?

I hypothesize that the failure dynamics observed when using continuous error measures may be related to the known behaviors and properties of those algorithms. These behaviors include algorithmic recursion, previous understandings about algorithmic reliance on specific operations, the location of algorithmic loops, and understandings about algorithmic space and time complexity. This hypothesis can be falsified if no discernible pattern relating to known algorithmic behaviors is found in algorithm failure dynamics data.

1.4.3 Can Failure Dynamics Analysis Improve Algorithmic Performance in Resource Constrained Environments?

Much of the analysis of efficiency in algorithms and computational systems assumes that enough resources will be available to provide a 'correct' result given algorithmic

correctness and then seeks algorithms that decrease the number of resources necessary to obtain that result. However, there are some situations where even the most efficient algorithms consume too many resources to allow for strictly-correct program solutions. This dissertation focuses on circumstances where there are not sufficient resources to perform every algorithmic operation correctly but where it may, instead, be possible to move resources to those computational steps with the greatest impact.

I hypothesize that a revealing analysis of failure dynamics within a particular algorithm should make it possible to leverage an economy of error to increase algorithm performance in a resource constrained environment. This hypothesis is falsified for some economy of error if we find conditions where a detailed failure dynamic analysis cannot be paired with an economy of error to improve algorithmic performance.

1.4.4 Which Operations Respond Well to This Method?

There are some operations and computational components that are vital for a program or system to produce partially correct outputs. Further, it seems that in the future advanced scheduling systems will be able to pull some operations out of the program stream and send them to less reliable hardware while ensuring that critical operations receive greater attention.

This dissertation hypothesizes that algorithms respond well to this method when it is applied to those operations that make up the bulk of the algorithm's run time, but where each are responsible for a small part of the program output. Examples include additions in multiplication, searches in fractal image compression, graph steps in graph search algorithms, ray traces in a ray tracer, backpropagation steps used for stochastic gradient descent, and comparisons in sorting algorithms.

Between these operations, I hypothesize that there must also be some *leverage*—operation failures must have a differential impact on the output based on when

and where they occur. A further hypothesis is that this leverage can be found when algorithm uses a mathematical or programmatic trick to increase its efficiency against some brute-force solution that is less efficient.

This hypothesis is falsified if algorithms without leverage, or that make use of brute force solutions, are impacted in some strong manner by the error economization procedure.

1.5 Procedures

In this work I focus on three traditional computer science problems: sorting, scalar multiplication, and matrix multiplication. For each problem I pick out a common computational step that is shared by all the algorithms in that problem set and inject faults into that step. I then measure the deviation between fault-injected and non-fault injected program runs to evaluate program failure criticality.

I then run an experiment where all computational operations subject to fault injection from step 1 are failed using an i.i.d. error model. Using the average output error from this i.i.d. experiment as a baseline result for failures on each algorithm and problem, I also run an experiment where I fail the least critical operations at double the failure rate while armoring the most important operations against any failures. I apply this experiment only to the multiplication algorithms. This is my error economization result.

Finally, using insights gained from the evaluation of program failure criticality on the sorting algorithms, I present a theory of robustness at scale that suggests a connection between algorithmic efficiency and algorithmic robustness.

1.6 Thesis Outline

In the next chapter I present background material concerning the techniques and ideas explored throughout this work, especially those exemplified by each of the concepts in Section 1.3. Chapter 3, building on work available in [1], presents a detailed description of the specific model and methods used throughout this work to answer the questions found in Section 1.4. Next, Chapters 4 and 5, based on work first explored in [1,2], present the results of empirical studies on criticality and error economization in both sorting and multiplication algorithms. Chapter 6, based on work first published in [3], is a general theoretical view of error and robustness through the specific lens of sorting algorithms that builds upon empirical observations made in Chapter 4. Finally, in Chapter 7 I consider how the concrete choices made in the development of this work limit its scope and present recommendations, speculations, and conclusions.

Chapter 2

Related Work

As discussed in Section 1.3 there are a number of concepts that touch on or underpin the work I present in this thesis. These include *criticality*, *fault tolerance*, *fault injection*, *quantified correctness*, *approximate computing*, *error economization*, and *theories of robustness*. Here I summarize work in each of these areas.

2.1 Criticality

Criticality analysis has a long history as a method for understanding failures in industries working with finite engineered machines. While this analysis is taken to be generalizable, its application is often limited to industries where safety is important, such as medicine, cyber-physical systems, and travel [36–39].

In these analyses a bottom-up approach that is similar to those found in this dissertation is often used in which known failure modes are assigned to engineered components and simulations of whole-system errors are produced by searching through a space of failures on each low-level component [36]. Such analyses have also been used to evaluate system susceptibility to malicious attacks, such as the transportation

network analysis found in [37] where the authors propose a criticality ratio measure that may be useful as an error measure for criticality evaluation in graph-based algorithms. Authors have also examined resource economies that use system criticality to balance real time operations [38, 39].

In each of these works criticality analysis is performed on projects that affect day-to-day living conditions. Even when these analysis are aimed at electronic compute systems, however, they are most likely to be developed for those systems that will have major 'real world' impact, such as the schedule balancing work for Boeing found in [46].

Many authors, however, move beyond criticality analysis to analyze operation, code, and system *importance* more generally. A related line of research in pure computing lies in *sensitivity evaluations* that often focus on the affects of system perturbations on whole system performance. In [47] the authors introduce time-based system perturbations which produce system performance degradation and then they evaluate mean time to recovery for the system as a sensitivity metric. Alternatively, in [48] the authors show how changes in system parameters such as micro-service failure and repair rates affect whole system performance metrics. However, sensitivity evaluations differ from criticality evaluation by looking at how failure events impact the *non-functional* behavior of a system.

Analysis of code importance and tendency to fail has also been used to aid designers in tracking vulnerabilities when introducing code edits [49–51]. These evaluations can be used to direct developer attention to the most important code changes, however, they do not focus on the effects or propagation of hardware failures all the way through the computation.

While criticality-focused work exists in the purely compute-oriented literature (e.g. [52]), it has been the tradition of compute-oriented engineering to focus on fault-tolerant compute paradigms. These paradigms aim to create perfect answers and

throw out any results that do not conform [10]. This has been possible because of the relatively low failure probabilities encountered when modern engineered systems are kept at a small scales and run within strict parameters of both resource consumption and guaranteed physical safety that may not be available in the future.

2.2 Fault Tolerance

Fault-tolerance techniques—the mainstay fallback of the computing world—rely on the idea that if a fault occurs it should be corrected, and if that is not possible the computation should be terminated. This paradigm, dedicated to the production of strictly correct results, can be found operating at both the hardware and software levels.

Examples of hardware oriented fault-tolerance techniques include [12–15]. In [12, 13], for example, a process known as *simultaneous and redundantly threaded processing* is used to check for transient hardware failures at the CPU level by running at least two identical threads and checking thread reads and writes to parity protected memory against each other. The authors of [14] improve on this process by scheduling threads such that certain latency concerns are minimized and execution can be guaranteed across different CPU cores for each thread. The authors of [15], alternatively, make use of out-of-order execution operations in super-scalar CPUs to check for correct execution against transient hardware failures.

Software-oriented fault-tolerance techniques, however, can be found in [16–19]. [16] discusses *SWIFT*, a tool that compiles code to add redundant instructions that take advantage of unused parallel compute resources to ensure successful control flow in programs on failure-prone hardware. In [17] the authors present a number of methods, implemented in software, to protect against data corruption in the face of single event upsets—electronic disturbances caused by hardware encounters with

radiation in space. A control flow analysis is used to create execution invariants that can be easily checked against real world software operation in [19].

All together these works focus on the use of n-modular redundancy through instruction replication and checking to guarantee deterministic execution. While some of the works overlap this dissertation through the use of i.i.d. failure models (e.g. [53, 54]), most works explore the effects of finite-bound failures, often one-off failures.

A further extension to the software fault tolerance framework occurs in [55–57]. These works focus on improving fault tolerance by allowing software authors to mark some code as more or less critical. Their methods are somewhat different from previous software fault tolerance techniques, and more similar to the work of this dissertation, since they focus on leveraging software developer's knowledge of code robustness and code *criticality* to redistribute compute resources towards protecting the more vulnerable code regions. However, this dissertation extends these research efforts by providing services that *discover* critical code operations through *space* and *time*.

This dissertation further differs from these works in that criticality explores what happens to computations when failures are *not* caught and corrected by fault-tolerant techniques. This is important for multiple reasons. Today, large scale systems often operate under the threat of silent data corruptions (SDC) [4]. The projected cost of maintaining determinism in exascale-class HPC machines is becoming increasingly prohibitive [20,58–62]. Further, authors are also pointing to a world where low energy chips may allow for a tradeoff between energy use and computational reliability [5, 25, 31, 63]

An important extension to the field of fault tolerance focuses on *fault injection*—the simulation of hardware and software failures that can then be used to test program behavior under faulty conditions. This empirical field is interesting because

it has produced techniques that can be used to study problems that lie both within, and outside, the fault tolerance paradigm.

2.3 Fault Injection

Fault injection is an important technique in the study of computational robustness, allowing experimenters to observe the behavior of systems subject to various kinds of hardware failures and software faults. A significant problem for this subfield lies in the difficulty of injecting hardware level failures into software systems. There has been a tradition of prototype fault-injection studies performed at the hardware level on built-for-purpose systems where both the software and hardware were created by the same manufacturer [64]. Today, however, systems are often built on off-the-shelf commodity hardware purchased from multiple producers. Under these circumstances injecting faults at the hardware level for study throughout the whole stack is difficult. One solution is virtual fault injection, which simulates the entire hardware stack with an emulator as proposed by [65, 66].

Fault injection, as a field, also focuses on how individual system-component bugs create faults in a broader computational system [67, 68]. An example of this is [69] a study of software-injected software faults which shows that some fault injectors are not as representative as they could be since regression tests are likely to catch the kinds of failures they propose as possible bugs. Representativeness studies, however, are not limited to software-level fault injectors on software-level faults. The authors of [70] for example, examine the representativeness of virtually-injected *hardware* faults using software-based simulators. Representativeness is an important subfield of fault-tolerance research that is not directly addressed by this dissertation, but which is addressed in future work.

Software-level bug-induced faults are also considered by [71]. The authors present

a framework that measures the impact of bugs in one software service on an entire system of software services. This is a common infrastructural paradigm for large-scale public-facing Internet organizations where potential conflicts may be caused when different versions of the same code base are running at the same time.

Altogether, these fault-injection techniques are used to extend fault-tolerance to Internet platforms through the creation of fault-prediction mechanisms—systems that use machine learning to predict the occurrence of faults before they happen. However, these systems require a great deal of fault data that is often not available. The authors of [72] propose a fault-injection technique that puts a simulated system in a faulty state which can then provide positive examples for fault prediction systems.

My dissertation both complements and critiques these fault-injection studies. For example, an interesting development in the fault-injection field has looked at the plausibility of an attacker *physically* attacking a system to upset its most vulnerable computational steps, with some degree of reliability such as [73–77]. My theoretical work complements this empirical fault-injection work by showing the existence of a sorting algorithm that is robust even in the face of worst-case malicious attacks under a rate-limited error model of failure at all scales.

Effectively, my criticality analysis is a kind of fault-injection framework for studying program robustness to failures. However, it is focused on fault-injection at the software level. Additionally, unlike most software-level injectors it is not focused on software bugs or the simulation of realistic hardware failures. Instead it focuses on analyzing the ways that an algorithm may naturally amplify or attenuate the impact of failures that occur at the level of program-visible operations, no matter what their actual path into program space. Further, since I do not perfectly simulate hardware failures, I can instead implement failure modes that explore how coordinated errors, below the object-function level, impact program performance in keeping with results that show that low-level failure coordination can produce greater error rates than

uncoordinated failures [78].

A final significant difference between my work and much of the fault-injection field lies in the way that I measure system performance. Many authors measure two kinds of values to evaluate fault-injected systems. The first include a series of non-functional measures that are designed to capture the overhead of the fault-injection framework and any fault-tolerant code that is tested using the framework. These overheads often come in the form of compute times, memory increases, and changes in network usage. The second set of measures used by most authors is that of expected system correctness, or the percentage of times that a fault is caught before it can have any functional impact on the computation's outputs.

My dissertation follows a different tradition that instead seeks to use quantified correctness measures. These are able to capture the degree to which a fault-impacted computation conforms to the correct output even when it deviates somewhat. That is, I seek to quantify correctness.

2.4 Quantified Correctness

While the fault-tolerance framework abhors any uncaught failure, there are still works within the framework that allow a failure to propagate all the way to program output with the goal of measuring the probability of strict correctness on computation output. For example, in [79] the authors measure full-system sensitivity to simulated *single-event upsets*—random bit flips caused by high-energy particle collisions within a computer. Alternatively, in [66] the authors find the probability that faults injected into a high performance computing application will cause either no change in the final output, an SDC, or cause a fatal crash. This tripartite evaluation lays the beginnings of quantified correctness. However, other authors have moved further.

Performability is a technique for evaluating whole computational systems with

many subcomponents which goes beyond even the tripartite no change/SDC/crash notion of systemic correctness. Each component of the system is given a functional strict correctness measure and/or a few non-functional measures to determine a score for that component. The whole-system performance is then evaluated by combining the scores of each of the subcomponents. Thus a system may degrade more or less gracefully from perfect performance to completely non-functional. Examples of performability metrics include job completion counts, link failures, output deviations, or increases and decreases in throughput [80, 81]. However, performability still relies on strict correctness measures to evaluate each subtask's functional performance. It could be extended so that subtasks have quantified error measures, however I have yet to find such efforts in the literature.

There is some research that envisions failures altering the output of a function. For example, the authors of [43] allow for program inputs that deviate from the correct input, while disallowing failures in program operations at runtime. Their method, called program continuity relies on showing that small changes in program input produce small changes in program output using some error measure. Using this method, they were able to show that bubble sort is continuous.

Alternatively, the selective reliability approach, discussed by [82], develops error bounds on computations that are divided into higher and lower reliability sections. Lower reliability sections are subjected to random hardware bit-flip faults that can modify the output of the whole computation. The present work in some ways complements that approach, seeking to identify computational steps most in need of high reliability. However, as previously stated, I focus on program-level failures, in comparison operations while sorting, or in the addition and bit-checking operations involved in multiplication, as opposed to hardware-level failures.

A paradigm of correctness sensitivity also complements works, such as [83] that seek to use oracles to search for failures in traditional software-engineering settings by further softening oracle requirements. Many evolutionary and machine-learning-

oriented engineering paradigms, such as *search-based software engineering* [84], may find that quantified correctness helps practitioners avoid search spaces with *flag-variable problems*—a situation where a software search space presents no signal of a problematic operation outside of a few unconnected points in the space that causes the triggering of a flag variable. With a quantified correctness measure, the spaces around such events may be easier to spot.

Overall, quantified correctness opens up new unexplored territory for computer scientists. If the goal isn't perfect computation every time, but is instead to reach for close results that degrade in some predictable manner as environmental conditions deteriorate, it becomes possible to find nice instances where computational *slack* [85]–the resources wasted on ensuring correct computation—can be turned into useful compute resources. It is also possible to create computations that can survive harsh compute environments that do not provide enough resources for traditional computing.

2.5 Approximate Algorithms and Computing

Approximate algorithms and computing are two closely related subfields that focus on obtaining useful results when it is not possible or feasible to obtain correct outputs [30]. The two techniques however, provide approximation at different levels. In approximate algorithms, individual operations provide results according to their *definition*, however the algorithm or individual operations are *defined* so that they provide only close-to-correct outputs. Alternatively, approximate computing employs algorithms and operations guaranteed to produce correct results if each individual operation performs according to its definition, however they are run on fallible hardware such that individual operations occasionally perform incorrectly.

Early approximate algorithms, such as in [41] focused on tasks that are still con-

sidered unfeasible, such as the maximum independent set problem. Approximation algorithms in this tradition focused on obtaining results that are near optimal, given some limitations on the query set. However, while many of these algorithms made use of randomized operations as in [86], the primary goal of authors lay in decreasing the use of compute resources in deterministic fault-tolerant environments. While randomized operations could be used to provide the search field necessary for successful approximation, the randomized operations were often either pure mathematical objects or the result of some psuedo-random deterministic process, with the randomness taken as part of the new approximate definition of the algorithm and its operations. That is, authors did not focus on how approximation algorithms could be improved through the use of actually-existing low-energy randomized switches at the hardware level.

An advance in this direction is the sub-field of *resilient algorithms*. A subclass of approximation algorithms that also lies within the field of approximate computing, resilient algorithms focus on providing approximately correct results on failure-prone hardware. For example, the authors of [87] transform deterministic algorithms, such as sorting, min-cut/max-flow, shortest distance into a gradient descent problem on a matrix with a different fitness function for each problem. Since gradient descent works even when the gradient is approximate, it is possible for the gradient descent problem to succeed even when running on a *stochastically correct processor*—a processor that is only guaranteed to return correct results some of the time.

While these methods rely on the mathematical stability of gradient descent algorithms in the face of occasional noise, they may not be suitable for all conceivable computations. Further, they don't provide an explanation for how computations interact with failures induced by hardware. Instead, they lift the fault tolerance framework to a new level, placing the whole computation in a space where failures are unlikely to affect the outcome while hiding failure dynamics. Nor is it certain that the gradient descent method will perform as efficiently as traditional algorithms, cre-

ating a problem where increased per-operation efficiency on reliability-constrained hardware is swamped by slow convergence time. For example, while the authors of [87] show decreased power usage for least squares problems, they don't show it for sorting.

An approximate computing approach that is much closer to the criticality techniques presented in this dissertation is called *Application Resilience Characterization* (ARC) [5], based on dynamic binary instrumentation [88], ARC supports approximate computing by helping programmers understand how their applications may function in failure-prone environments. This code analysis technique determines which code is *sensitive* or *resilient* and is complemented by hardware/software schemes such as Flikker [32] that allow programmers to use partitioned code (called *critical* or *non-critical* in the Flikker scheme) on hardware with varying energy usage and reliability characteristics. Overall, the authors of ARC show that many algorithms can stand either a low rate of arbitrarily bad failures, or a high rate of low impact failures. However, both Flikker and ARC have focused on algorithms that are inherently robust, such as machine learning algorithms and video processing computations. Perfect correctness is rarely *expected* in such environments.

These complement my work—which focuses on traditionally deterministic sorting and multiplication algorithms—by providing a quick method for determining which operations qualify for the constrained reliability allocation problem. However, these methods analyze code-level operation failures, assigning each line to one of the tripartite no change/SDC/crash categorizations. In my work, I focus on time-bucketed quantitative evaluation of the impact of failures on program output which provides a richer understanding of how failures interact with computational outputs. This understanding is useful because it can be leveraged to improve the performance of computations running in reliability-constrained environments through the usage of economies of error.

2.6 Error Economization

Error economization is the process of establishing a common relationship between all possible failures such that one failure can be avoided by increasing the resources spent on that failure and decreasing the resources spent on some other failure. When some knowledge of the final impact of a failure on computational *error* is also understood, it becomes possible to leverage this economization in a manner that redirects resources from trivial failures to more important ones.

Some of the works that I have already discussed implicitly contain a kind of economization of failure. For example, in [38] the authors present a scheduling algorithm for cyber-physical systems that allows high criticality operations to trump low criticality operations when a system encounters a spike in work load. This can be viewed through a quantified correctness lens as an increase in low criticality operation failures and a decrease in high criticality operation failures. The authors of [39] present a similar balancing of resources between critical and non-critical network communications in a medical network and, in some sense, the entire field of network quality of service (QoS) can be seen as a kind of economy of error where low criticality communiques are stalled or dropped for the benefit of higher criticality ones.

Some of the work closest to mine on the error economization front can be found in [33]. There, the authors present a concrete hardware platform for a camera in which they provide quality-energy curves that show how energy usage in various hardware components relates to total system output quality along a Pareto front of best quality for the energy buck. However, the process of providing this curve requires detailed hardware knowledge specific to the particular platform they were working with. An advantage of this dissertation is the lack of such a requirement.

2.7 Theories of Robustness

Looking beyond the empirical techniques and frameworks that have been used to analyze computational robustness, this dissertation also seeks to explore the theoretical underpinnings of computational robustness research. Concretely, a large amount of my work has focused on sorting algorithms. While I have presented some works that focus on solving the sorting problem in the face of failures (e.g. [42,53]), there is considerable work under the fault-tolerance paradigm that explores the theory of robust sorting networks and techniques when dealing with faulty comparators. In [89] the authors present a sorting network such that there are only n comparator pairs which produce a faulty output when both comparators fail. Interestingly, [87] develops robust floating point numerical problem solvers for several exact algorithms, including sorting. They demonstrate correct sorting on small lists with even as many as half the floating point operations failing.

Moving beyond theories of robustness in the fault-tolerance paradigm, I have focused on two different scenarios for the distribution and modeling of failure—correlated worst-case models that can be used to simulate malicious attacks and uncorrelated i.i.d. failure models. In both scenarios I have used rate-limited models where only some *percentage* of all computational operations may be corrupted.

The rate-limited worst-case model I use is very similar to the model found in [44]. In this model an attacker is allowed to modify up to δ memory locations and during their analysis they often transform this value into a number of corruptions per unit time σ. To deal with these modifications, the authors provide resilient sorting algorithms that can provide outputs that are provably 'k-resilient'—that have at most k items out of place. While this error model is very similar to that found in Chapter 5 for worst-case algorithmic robustness, one weakness of their proof is that the allowed proportion of total fallible memory locations *shrinks* as the size of the algorithm grows. Specifically, their FAST sorting algorithm only allows $\delta = O((n$

$\log n)^{1/3})$ modified operations even though the run time is $O(n \log n + \delta^3)$.

There is a long tradition of using i.i.d. models of computational failure. In [53] the authors present a sorting network that can sort an input list with high probability as long as each comparator has an i.i.d. failure probability less than 1/2. The authors of [54] use an i.i.d. failure model on connections between nodes in a grid computing environment. Further, works I have already discussed (e.g. [5]), also tend to default towards i.i.d. models when no strong argument for some other model presents itself. This is reasonable, considering that i.i.d. models exist to handle situations where there is a lack of information concerning the correlations of various events as is often the case with many physical processes.

However, there is a subtle difference between my model and most previous i.i.d. failure models. When these models are rigorously applied at the level of hardware, many failures are caught by statistical fault-tolerance techniques before they can ever impact actual program outputs. Therefore, failures such as these tend to impact programs only if they are correlated. My work, however, injects failures into applications by directly modifying program operations, skipping detailed hardware simulations. This is effectively the same as asking what will happen *given that* failures are sufficiently correlated to impact program operations. In this way, my i.i.d. failure model at the level of *program operations* is also a model of correlated failure at the level of *hardware*.

Finally, my theoretical work on the $lg(n)$ growth of operations in i.i.d. resistant scalably robust algorithms complements empirical results from Fiala [90] showing that scalable determinism requires ever more reliable hardware components. Avoiding this outcome requires a more sophisticated model of failure and error.

Chapter 3

The Model

3.1 Author Contribution Statement

I am the lead author of the paper underlying this chapter under the supervision of Dave Ackley (Associate Professor Emeritus, UNM Computer Science). I outlined, developed, and wrote the description of the model presented in this section. Dr. Ackley aided in editing the chapter and developed Figure 3.1 in consultation with me.

3.2 Publication Notes

Citation: T. B. Jones and D. H. Ackley, "Damage reduction via white-box failure shaping," in *International Symposium on Search Based Software Engineering*, pp. 213-228, Springer, 2018.

Editors: Thelma Elita Colanzi Lopes and Phil McMinn

Received: March 29, 2018

Accepted: June 6, 2018

Published: September 8, 2018

Acknowledgment: Adapted by permission from Springer Nature Customer Service Centre GmbH: Springer Nature *International Symposium on Search Based Software Engineering* “Damage Reduction via White-Box Failure Shaping,” T. B. Jones and D. H. Ackley, © 2018

Formatting: Some parts of this chapter are derived directly from the original text, especially Figure 3.1, however many parts are new. Figures from the original text are used with permission. Figures are mostly unchanged, though they have been modified to fit this dissertation.

Data Availability Statement: Data is available at
`https://github.com/ThomasBJones2/Robustness_Dissertation_Data`.

Funding: No funding was provided for this work.

Competing Interests: The authors have declared that no competing interests exist.

Supporting Information: Supporting information for this chapter can be found in Chapter 5.

3.3 Introduction

There are many potential relationships between failed operations, correct operations, and the output of an algorithm. A failed operation at one point in an algorithm on one particular input may have a very different relationship to the output of the algorithm than an operation even a small distance away. A complete description of failure dynamics may be too complex to model.

Rather, this dissertation seeks a rough understanding of failure dynamics on a limited set of relationships between operation failures and algorithm output. The perspective used by this dissertation is that of a *criticality assessment*—a method for measuring the amount of additional error in algorithm output created by a failure on a single operation at a specific location in the algorithm, all other things being equal.

Once this assessment is performed, in some studies I also pair the results with an economy of error and a simulated method of resource redistribution to show that criticality results can be used to improve algorithmic performance. These economies of error restrict the algorithm to an insufficient amount of resources necessary for correct execution. I show the usefulness of the criticality assessment method by evaluating the performance of the algorithms with, and without, economic rebalancing.

In this chapter I discuss criticality assessments in greater detail in Section 3.4. In Section 3.5 I explore how quantified correctness is used in the form of error measures to evaluate program performance throughout this dissertation. Next, in Section 3.6 I explore the use of space and time in the criticality assessment method and in Section 3.7 I present a formal, mathematical definition for criticality as used throughout this work. Some of the studies in this dissertation were produced through the use of a general-purpose tool, *Criticality Explorer*, that I wrote and which is explained in Section 3.8. Finally, in Section 3.9, I explore the technical specifics of using criticality assessment results to solve the constrained reliability allocation problem.

3.4 Criticality Assessments

In this dissertation I present a method of assessing operation failure criticality that is based on measuring the outcome of Monte Carlo simulations of algorithms run with faulty operations. These simulations require an *input generator*, an *error measure*,

and some notion of an operation's *failure mode*, or behavior when a failure occurs during its execution.

An input generator produces random inputs to the algorithm following some statistical rule or distribution. The input generator has two requirements:

1. It must provide well formed inputs for the program—namely, it must provide lists for sorting, numbers for addition, graphs for max-flow/min-cut, etc.

2. Inputs produced by the input generator must be random.

Outside of these, there is no further requirement placed on the input generator *in principle* by this dissertation.

Specific, concrete error measures used in this dissertation will be covered in more detail in Chapters 4 and 5. For now it suffices to say than an error measure accepts two outputs from some algorithm under examination and then provides an evaluation of the distance between these two outputs. In order to perform a criticality assessment, one of the outputs is produced by a run of the algorithm in which some failure has been simulated while the other output, generally called the "correct" output or "more correct" output, will be produced by a run of the algorithm without that particular failure. The difference, as calculated by the error measure, is the operational instance criticality.

An operation's *failure mode* describes how the operation behaves during a failure. This allows us to ask what would happen if a failure were to occur, unchecked, at some given location or moment within the algorithm. Failure modes are meant to abstract correlated errors which may occur in the hardware and software stacks beneath an algorithm so that it becomes possible to ask how the algorithm *itself* behaves when it encounters a particular computational failure without aid from the hardware.

An example failure mode could be written for numerical comparisons in which the outcome of some comparison between two numbers returns the opposite of the correct result of the comparison. Given an operation $x < y$ which returns true if x is less than y or false otherwise, an acceptable failure mode for the operation might return $x >= y$. When executed on $x = 3$ and $y = 4$ *in its failure mode* this operation $3 < 4$ would return *false* even though in its *correct mode* it would return true.

Another failure mode may return $\texttt{rand}(True, False)$, a coin-flip result instead of the correct result. In this failure mode the operation $3 < 4$ would return *True* half the time and *False* half the time. Either of these would be an acceptable failure mode. All that is required is that the failure mode accept the same inputs as the operation in question, plus a random number generator, and return an output of the same *type* as the original operation without throwing an exception or warning the algorithm in some other way.

So far, I've discussed the components used in a criticality analysis. In keeping with the approximate design paradigm, faults in program operations are allowed to percolate into program output. Once in the output they produce errors which can then be measured to produce a failure *criticality*—by performing one run of the algorithm with that particular failure and another run without a failure and then taking the difference in the output error between the two runs.

This difference is evaluated through error measures. However, a major hypothesis of this work is that error measures that evaluate whether a computation has been performed correctly are insufficient to produce interesting failure dynamic results. Instead we need quantified error measures that score algorithm outputs on a 'curve' and that give 'partial credit'.

3.5 Error Measures

For the sake of capturing each failure's criticality, an error measure is necessary to evaluate the output quality of the computation with and without the failure. The error measure $E(O_e, O_c)$, takes outputs from an errorful run of the computation, O_e and compares it to a correct or 'less errorful' run of the computation O_c. In this work, I present both normalized and unnormalized error measures with normalized error measures set to return values in [0..1] and unnormalized error measures set to return unbounded *positive* error. A normalized error measure need not be completely continuous in [0..1], however a good measure should attempt to hit as many values in that range as possible.

For example, the $L2$ error from linear algebra would make a good a quantified correctness measure. This measure is the sum of the squared distances between all elements in a vector and some reference vector of the same size. However, it would also be unnormalized. To normalize the error measure, one might pick a large $L2$ error cutoff, and send all $L2$ errors above that value to 1 while dividing all other output errors by the cutoff value.

In Chapters 4 and 5 I will present specific error measures for sorting, scalar multiplication, and matrix multiplication.

Using these error measures, for small computations it may be possible to calculate every specific failure's criticality exactly, with each failure being identified as a single flaw in the total state of the program throughout its entire run. But many algorithms have a state space that is combinatoric and so algorithms of even moderate size cannot be assessed in a reasonable time frame.

Instead, this study uses Monte Carlo sampling to estimate the error measure values with and without the failure. Rather than consider each failure from each different input separately, failures are bucketed together by their characteristics and

an average increase in error is calculated for each bucket. A novel advance of this dissertation is the use of the failure's *location* in the algorithm to calculate these averages.

3.6 Mathematical Abstractions of Space and Time

Locating a failure within a computational system is a trickier prospect than it might first appear. It is important to remember that computations, for all their virtuality, are first physical processes than involve the movements of energy and matter through real objects, no matter how small those movements may be. Further, programs impose a certain logic on the geometry of the hardware they sit upon. This event cannot occur before that event, this data must be close to that data and they must combine with each other within a certain number of seconds.

However, a too-strong insistence on examining computations at the level of electrons and transistors can also leave us without a full understanding of the virtual spaces that exist within a computation at higher levels. Further, for some analyses, however incomplete they may be, it becomes unimportant whether the computation is occurring on a head moving up and down a piece of tape, or in a central processor, or across a series of cores distributed throughout a large system. And, therefore, many algorithmic analyses use very abstract notions of space and time. In these analyses all points in space touch each other and time is a single universal clock.

This work proposes an analysis of space and time that lies between the physical and the abstract. For the purposes of this work, each operation failure occurs at some *failure point*, $K = (Op, i)$. This failure point is is defined by an operation, Op, and an *execution index*, i, that counts the number of times the operation has so far been executed in the computation. The operation Op, is any function in the code

that accepts program state, mutates it, and returns a new program state. Operations can be simple, like addition, or complicated, like a call to a sorting function. An example operation failure may be located at $K = (43 : (int) + (int), 3)$. Here, K is equal to the third time the operation $43 : (int) + (int)$ - a sum of two ints on line 43 - is executed in the program. Notice, that in a serial deterministic algorithm with a single failure mode, failure points are unique to each failure.

For a failure point, the operation is an abstraction of space while the execution index is an abstraction of time. A particular operation in the code often occurs in the same physical location in a computational system. Further, program states that have very similar impacts on program output often move together, through a given operation, within a very small window of possible execution indices.

Operations must also be packaged with a failure mode, as described, that perform an incorrect mutation of program state, from the perspective of the program. Operations in this work are allowed to have only a single failure mode, however future analyses may employ more. This binary modality does allow for a simple failure mode notation, however. In this work, K^* notes that the operation at K employed its failure mode while K means it ran its correct mode.

An additional requirement of this work is that failure modes not mutate large segments of program state. Instead, they only affect the return value of an operation or the program state that the operation mutates in the last few moments of its execution. This helps strengthen the notion of an operation as an abstraction of space as the operation's failure is unable to affect all program state unless it first affects the behavior of other operations.

Using this spatio-temporal paradigm we can then develop more completely the idea of a failure's criticality when it is bucketed by location. Rather than consider the additional error due to all failures, or due to each failure individually, we can begin to ask questions about the additional error due to a failure at a particular

failure point.

3.7 Criticality Defined

A rough definition of criticality, as the extra error due to a single failure, has already been given. However determining the extra error due to every failure under each distinct circumstance is infeasible for large programs and does not produce a parsimonious description of failure criticality. Instead, I have presented a spatio-temporal representation that ties each failure to a particular unique location in a computational process. It is now possible, using this representation, to provide a more precise definition of criticality as it is used in this work while also accounting for the interaction between a failure and background failures that might additionally occur during the execution of a program.

However, failures can have strange behavior. Sometimes, a failure may only impact the output of a program given that other failures have also occurred [45]. It is possible that a certain amount of background noise is necessary before a failure impacts the output, as occurs under n-modular redundancy. In my definition of failure criticality I use the representation outlined in the previous section to design reproducible experiments measuring the additional error due to a single failure under many different conditions—both those with, and without, background failures.

Reproducible criticality experiments are performed through the use of a *failure pattern*—a record of when and where in the computation each failure occurs. Given a program description of length M and an expected max run time of T, each of the MT failure points is marked with the star-notation used above to produce a failure pattern. So, for example, a program with three occurrences of a single operation K might have a failure pattern: $(K, 0), (K^*, 1), (K, 2)$ that shows a failure on only the second instance of operation K.

We also use the star notation to mark the difference between a failure pattern *with* a particular failure and a failure pattern *without* a particular failure. If f is a failure pattern, I use the notation $f \wedge K^*$ to mean the failure pattern with failure point K in its failure mode and $f \wedge K$ to mean failure pattern f with K in its correct mode. Finally, to ensure the reproducibility of this operation, if the computation overruns its expected run time T it is marked as returning some maximum or unbound error, or the result is thrown out and recorded as a program time overrun, depending on the error score.

Since each failure point's criticality is meant to account for many possible failures occurring in the presence of many possible distributions of background failures, it is not enough to speak of a single failure pattern at a time. Instead, I use F to represent a *failure pattern distribution* or probability distribution over the set of all possible failure patterns of size MT.

There are many different potential ways to select the failure pattern distribution F. In general, I will consider two failure pattern distributions. The first is F^0—a distribution where all failure points operate in the correct mode. The second is F^ϵ—a distribution in which failure pattern probabilities are equal to their probability in an i.i.d. failure model with a failure rate of ϵ. Since failure patterns are drawn from these distributions I use $f \sim F$ to note that a failure pattern f has been drawn from the distribution F and that it is weighted by its probability of occurrence in F.

Since I am dealing with computations run under many different failure patterns, I adopt the convention C_f to mean computation C running with failure pattern f. Thus C_f is a simulation of computation C when specific operational instances, as defined by f, have been forced to fail. $C_f(i)$ is the result produced when computation C is run on input i over fault pattern f.

Using this failure pattern and the locality notation it is now possible to write a reproducible and parsimonious notion of failure criticality as the additional error due

to a single failure, as presented in Equation 3.1.

$$Crit_{C,F,I,E}(K) = avg_{\forall i \in I, f \sim F} E(C_{f \wedge K^*}(i), C_{f \wedge K}(i)) \tag{3.1}$$

This definition integrates all of the components so far discussed into a single notion of criticality that is oriented towards an understanding of failures as being in relation to

1. the programs in which they occur, captured by C,
2. input distributions captured by I,
3. failure pattern distributions captured by F, and
4. error measurements captured by E.

This understanding emphasizes the spatial and temporal nature of failures by placing criticality measurements in a field of failure points that are abstracted from physical space and time. However, that same abstraction also allows the concept to focus on the failure properties of specific high level algorithms when failures impact those algorithm's performance by escaping into the algorithm's compute space—as can occur when failures are coordinated or computation economies are too tight for strictly correct execution.

In the next section I present *Criticality Explorer*—a Java- and AspectJ-based [91, 92] tool I developed to perform criticality assessments on tooled algorithms. It can be found at `https://github.com/ThomasBJones2/CriticalityExplorer`. *Criticality Explorer* also performs an economic assessment of algorithm performance when criticality assessments are used to direct robustification resources towards the most important computational steps. I will also note some distinctions that lie between the theory of criticality outlined above and its practical implementation as embodied in *Criticality Explorer*.

3.8 Criticality Explorer

To measure failure point criticalities, *Criticality Explorer* requires an *input generator*, an *error measure*, and a set of *failure interfaces* for each algorithm assessed. Inputs are drawn from the input generator and outputs are assessed with the error measure while the failure interfaces take on the role of the failure modes from the previous section. See Figure 3.1 (page 46) for a visual overview of the experimental process used by *Criticality Explorer*.

In *Criticality Explorer* the input generator is handled by an *input object* which conforms to an input object Java interface. Input objects must have a `randomize` method that acts as the input generator to an *experiment object*. These experiment objects run the experiment program and provide a `score` method that evaluates the error measure difference between an errorful and a less errorful—sometimes called 'correct'—experiment object.

In *Criticality Explorer*, *failure interfaces* operate at the level of Java methods chosen by the user. In addition to the correct method code, the user provides an alternate *failure method*. This method has the same signature as the original Java method, except that it accepts an added parameter, a random number generator, to be used to simulate underlying failures in the stack beneath the method. Failure interfaces in the experiment object are annotated with '`@Randomize`' and require two implementations—the correct implementation, and a failure implementation—to function.

The failure interfaces identify a spatiotemporal set of *failure points* bucketed at some space-time granularity. These buckets are locations uniquely defined by their associated failure interface and an invocation count on that failure interface. *Criticality Explorer* assess the *criticality* of a failure point by estimating the expected degree of damage that failure point would cause, assuming all else is equal. Given a set of failure interfaces, I call the criticality distribution produced by the assessment

process a *failure shape*.

Failure interfaces could be defined at higher, lower, or just other levels of abstraction but I see five principal advantages to method-level failure interfaces:

1. In a direct hardware realization, the data paths of multiple instances of an object method often pass through similar or even the exact same circuits. Method-level failures may thus provide increased abstraction while still approximating important spatial failures of real hardware.

2. Method-level failure interfaces are flexible. Beyond the SDCs and energy economization failures that inspire this dissertation, they can also represent software bugs or failures in distributed computations. *Criticality Explorer* can be used to analyze a wide range of hardware and software failures.

3. Many robustness engineering methods are designed to compensate for independent, identically distributed failures. I too consider i.i.d. failures, but method-level interfaces can also model higher-order *coordinated failures* arising from deep within the computational stack. Specifically, failure interfaces written on major, central methods, can simulate SDC errors that have percolated through the stack to become visible to the user.

4. Since most modern languages treat methods as (nearly) first class objects, it is easy for software engineers to understand and implement failure interfaces at the method level.

5. Method-level interfaces conform to the intuition that SDCs occur when objects uphold their interfaces but violate their contracts.

Criticality Explorer records the number of invocations on each method-level failure interface to generate the failure point field. A small number of exploratory runs at tested input sizes and ϵ failure rate are performed in order to find each failure

interfaces maximum invocation count. Using this information *Criticality Explorer* then evaluates each failure point criticality through Monte Carlo simulation. A random input is drawn using `randomize` and an experiment, failed only at the given failure point, is compared with a failure-free run of the algorithm on the same input. The error in their outputs is evaluated using the `score` method.

Criticality Explorer can take advantage of AWS lambda to perform many criticality assessments at once. This makes it possible to quickly assess failure point criticalities at multiple input scales.

Because failed operations can cause unexpected behavior, *Criticality Explorer* automatically catches and records any exceptions produced by experiment code at run time. It also automatically terminates experiments after a hard-coded two-minute time limit and records the termination as a runtime error. This is similar to, but not exactly the same as, the maximum operation count from the previous section.

Another difference lies in the way that failure patterns are implemented in *Criticality Explorer*. Rather than using failure patterns, as described in Section 3.7, *Criticality Explorer* instead uses a random number generator with a fixed random seed to choose which operations' failure methods are triggered and the random numbers used to generate simulated failures inside a failure method at runtime. Given that algorithms are otherwise deterministic, this guarantees the same program behavior given the same input, and error measure.

When measuring failure point criticality, the algorithm must be run with the operation forced to fail and forced to succeed as discussed in the previous section. When the failed operation is forced to fail, however, it is possible that random numbers from the generator will be consumed, which means that the failures *after* the forced operation are not guaranteed to be the same in *Criticality Explorer*. This is the single major difference between *Criticality Explorer* and the original criticality

model [2], however there is a justification for this difference. Every failure tends to change the meaning of the operations that occur after it. It is important that the operations before a failure have the same behavior when measuring criticality, but there is no significant reason to believe that the computational paths after the failed operation need to be exactly the same for a successful measurement.

After criticality assessment, *Criticality Explorer* then performs three additional experiments, discussed in the next section. The first measures experiment code output error with an i.i.d. failure model with failure rate, ϵ, in $[0, 0.1]$. The experiment code is then *failure shaped*—reliability resources are redistributed from trivial to critical operations—according to the criticality assessment results. Finally, a proxy economization experiment that uses some failure points' criticalities as stand-ins for others is also available within *Criticality Explorer*. As I will show, this proxy method has the benefit of shrinking the criticality search space, providing improved performance at a lower cost.

3.9 Failure Shaping

Failure shaping is designed to deal with reliability budgets that are insufficient for strictly correct execution. This technique employs an *economy of failure* for each fallible method in a computation. It also makes use of a failure rate $\epsilon \in [0, 1]$ such that roughly ϵ of the failure points generated at runtime *will* fail. This is novel from a *fault-tolerant/intolerant* perspective since a budget sufficient to provide correct results is often presumed.

As a baseline *Criticality Explorer*'s economy of failure presumes a non-zero i.i.d. failure model at each failure point. Then, *Criticality Explorer* shapes the failures: the *least* critical failure points—those with a criticality below the median criticality—are adjusted to 2ϵ. Alternatively, the *most* important coordinates are given a failure

rate of 0. This keeps the total failure rate over the whole computation at ϵ. This failure redistribution scheme fits into a concept like *triage* where operations can either be lost (less than median criticality) or salvageable (greater than median criticality). The operations assumed to be failure-free form the 'no treatment necessary' category traditionally found in triage processes.

This stylized and simplified economy of failure is oblivious to whatever actual underlying mechanisms are used to shape failures within the system. I explore the effects of failure shaping without proposing a complete, concrete, failure shaping technology. However, relationships between power or energy and failure rate presented in [63, 85], for example, give me hope that economizations like this may be realizable in a few years.

However, the current method of assessing operation criticality, even with average-case location bucketing and optimizations using AWS lambda, is still very slow and does not scale well. One method to get around these problems is the use of proxy criticalities to shape operation failures. Under a normal economization each failure point receives resources determined by its criticality performance. If it has high criticality it receives a great deal of resources, otherwise it receives far fewer resources. When making use of a proxy criticality, however, each failure point's resource budget is *not* determined by its criticality. Instead, the criticality of some proxy operation is instead used to determine which operations should receive robustness resources.

An example of this proxy criticality method might crop up in a program that is subject to bitwise failures during an addition operation. A single addition operation on two numbers of size N could have, depending on the summation implementation, $3N$ such operations. While we may measure the criticality for every single bitwise failure point in an algorithm, the proxy criticality path instead calls for the creation of a failure interface for the addition operation as a whole. Then we might assign resources to the bitwise operations internal to the summation operation, based off the criticality performance of the total *summation* failure point they are a part of.

Using these methods—criticality assessments, average case error rates, failure shaping, and proxy criticalities—it is possible to uncover a new understanding of the failure dynamics of deterministic algorithms. In the next chapter I describe how these methods can be used to explore the failure dynamics of sorting algorithms in particular.

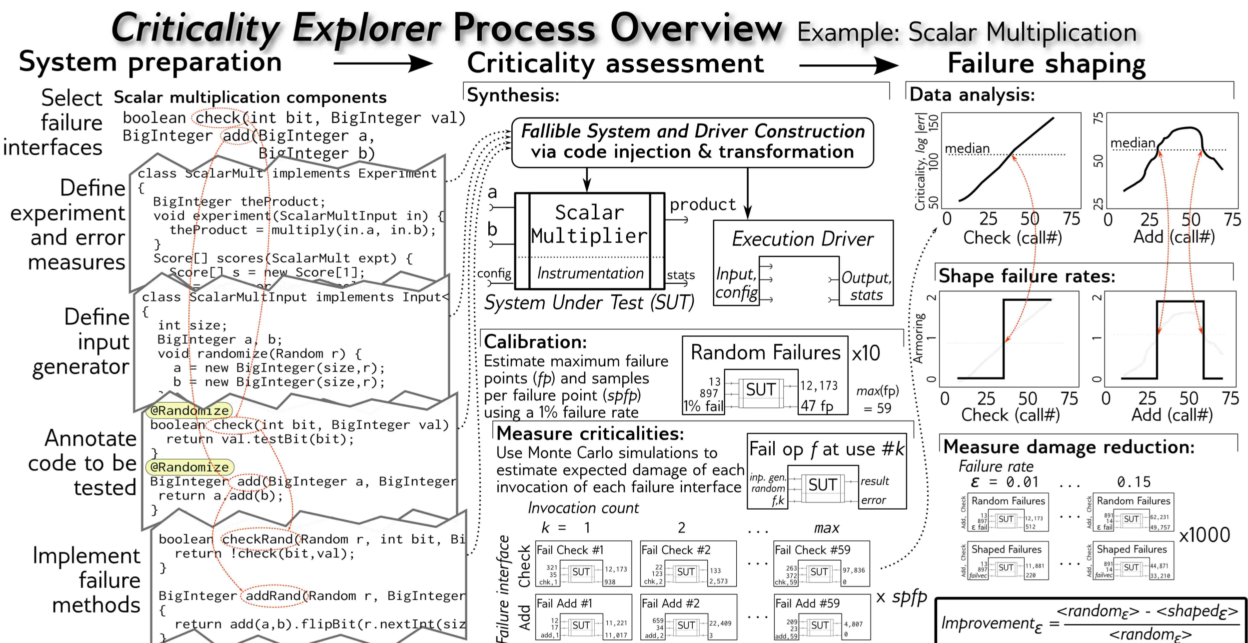


Figure 3.1: **Criticality Assessment and Failure Shaping Overview.** *Criticality Explorer* takes a prepared algorithm with error measures, input generator, and annotated method level failure interfaces shown on the left. After a calibration step to evaluate the maximum failure point count, and samples per failure point, the tool then estimates failure point criticalities through a Monte Carlo simulation. Each sample produces the output of a single run with, and without, a failure injected on the a given failure point and scores the error between the two runs. The errorful inputs, outputs, and absolute difference score can be found in the bottom center of the figure. Failures are then shaped using the median criticality and error scores are produced for average i.i.d. as well as shaped failures. See Section 3.8 for details. Figure reprinted from [1] with permission.

Chapter 4

Sorting Algorithm Criticality

4.1 Author Contribution Statement

I am the lead author of the paper underlying this chapter under the supervision of Dave Ackley (Associate Professor Emeritus, UNM Computer Science). I outlined, developed, and wrote the experiments that provided the data for this chapter. I also outlined, developed, and wrote the contents of this chapter, developed appropriate figures, and formatted the chapter. Dr. Ackley aided in the editing of this chapter and in the development of figures for this chapter.

4.2 Publication Notes

Citation: T. B. Jones and D. H. Ackley, "Comparison criticality in sorting algorithms," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (DSN), pp. 726-731, IEEE, 2014.

Editor: Juan E. Guerrero

Received: March 7, 2014

Accepted: April 11, 2014

Published: June 23, 2014

Copyright: © 2014 IEEE

Formatting: Substantial parts of this chapter are derived directly from the original text, however some parts have been edited for clarity or modified to fit with this dissertation. Figures are mostly unchanged, though they have been modified to fit with this dissertation.

Data Availability Statement: Data is available at
`https://github.com/ThomasBJones2/Robustness_Dissertation_Data`.

Funding: No funding was provided for this work.

Competing Interests: The authors have declared that no competing interests exist.

Supporting Information: All pertinent information in from the original work is contained in this chapter.

4.3 Introduction

The first study I present concerns the criticality of comparison operations in comparison-based sorting algorithms. In this study:

- I observe the criticality behavior of three comparison-based sorting algorithms: quicksort [93], merge sort [34], and bubble sort. A note: While many traditional bubble sort algorithms encounter half the list on the average pass (e.g. [94]), my algorithm encounters the full list on every pass and so I call it *full bubble sort*.

Because this sorting algorithm performs more comparisons, it has a robustness advantage compared to the usual bubble sort algorithm.

- The program input is a random permutation of the numbers 0..51, modeling a shuffled deck of cards.

- I use normalized presortedness measures as my error measures. The normalization ensures that each measure maps any permutation of the input data into a scalar value from 0.0 meaning “perfectly sorted” to 1.0 meaning “maximally unsorted.”

- I consider only failures in the pair-wise sorting comparisons. Such failures fit naturally into my adopted presortedness measures, but they are only one of many possibilities. In particular, I presume the data items are never corrupted. Discussion of the presortedness-based error measures used in this study can be found in Section 4.3.1.

- The failure mode of the comparisons is equal to the opposite of the result they would normally return: $3 < 4$ returns $True$ so a failed $3 < 4$ will return $False$. Alternatively, a randomized result could also work, however, its impact would likely lead to a measurement of criticalities with *half* the value as those seen in this study.

- As with all other studies in this thesis, whenever I draw from a failure pattern distribution *other than* F^0—the distribution will be drawn from an i.i.d. error model at a rate of ϵ.

The results in this chapter predate the development of *Criticality Explorer*, and each algorithm under consideration was manually tooled to accept both an input list and a precomputed failure pattern. The program then checked the failure pattern to set comparison operation modes. However, the basic underlying method of criticality

measurement used in this study is very similar to the more developed method used by *Criticality Explorer*.

4.3.1 Measuring Sortedness

To investigate sorting as an example, we must confront the question of what "'sort of' sorted" might mean. Fortunately, sorting is an extremely well-studied topic, and researchers have defined a variety of *presortedness measures*— [34] is one survey— that quantify the notion of 'partially correct sorting'. As their name would suggest, these measures have traditionally been used to measure a list's 'presortedness'—its degree of disorder before sorting—but they are also useful as output quality measures on a fallible sorting algorithm.

Existing presortedness measures include *inversions error*—the number of items immediately preceding a smaller item, and *max displacement*—the maximum distance any item must be moved to reach its correct position. In this paper, we explore those measures, as well as all-or-none *strict correctness*, and a measure called *Spearman's footrule error* which I have also called *positional error*. *Strict correctness*, is 0 if the output is sorted and 1 otherwise. Definitions for the other three error measures—*normalized inversions error* [34], *normalized max displacement* [34], and *normalized Spearman's footrule error* [95]—appear below in Equations 4.1, 4.2, and 4.3, respectively.

In those equations $L(i)$ is the position of item i in the output list L, while $L[i]$ is the inverse operation: The value of the ith item in list L. L_c is the output of a correctly sorted list. Since we sort lists of distinct numbers from 0 to $N-1$ the ith item in a sorted list is equal to $L_c(i)$. Note that each error measure is normalized to $[0, 1]$ by dividing by the maximum possible value of that error measure.

$$\mathrm{Inv}(L, L_c) = \frac{\sum_{i=0}^{N-2} \frac{L[i]-L[i+1]}{|L[i]-L[i+1]|} + 1}{2N-2} \tag{4.1}$$

$$\mathrm{MaxDis}(L, L_c) = \frac{\max_{i=0}^{N-1} |L_c(i) - L(i)|}{N-1} \tag{4.2}$$

$$\mathrm{SFE}(L) = \frac{\sum_{i=0}^{N-1} |L_c(i) - L(i)|}{\mathrm{SFE}(N)} \tag{4.3}$$

The normalization factor $\mathrm{SFE}(N)$—the maximum Spearman's footrule error for an input list of size N—is equal to the Spearman's footrule error when a list is reverse sorted:

$$\begin{aligned} \mathrm{SFE}(N) &= \sum_{i=0}^{N-1} |N - (2L_c(i)) - 1| \\ &= 2\left\lfloor \left(\frac{N}{2}\right)^2 \right\rfloor \end{aligned} \tag{4.4}$$

4.4 Results

The criticality for a failure at each comparison index was obtained by taking a sample of 1000 failure pattern-input pairs for each comparison in the algorithm. Inputs were randomly generated so that each list item had a uniform probability of occurring in any location in the list. Failure-patterns were sampled from a binomial distribution set to produce *true* bits at rates of 0%, 10%, and 20% so that comparisons not under consideration would fail at a consistent i.i.d. *background failure rate*[1].

To test the first half of Hypothesis 1.4.1 on sorting algorithms, I first measured each of the algorithms using strict correctness error. The results from this test can be

[1]Note that as we use it here a *background* failure rate of 0% means there are no failures *other than* the one failure being induced in the comparison under consideration.

found in Figure 4.1 (page 54). In this figure, the criticality of comparison operations at a 0% background failure rate is 1 nearly everywhere for the two algorithms that run in $O(n \lg n)$ time. A failure on any operation leads to a judgment of complete failure for the program. Alternatively, at higher background failure rates no operation seems to have any criticality—all the damage has already been done. $O(n^2)$ bubble sort, however, is anomalous. Most operations have no immediate impact on output and instead only the last n operations seem to have an impact at 0% background failure rate. Otherwise, at higher failure rates the pattern that shows no criticality holds.

There is an additional anomalous pattern towards the end of each algorithm where the final few operations did present with a criticality between 0 and 1. However this measurement reflects the probability of the algorithm reaching these final operations *at all* rather than some diminished impact they might individually have on output *given that* they have been reached.

In Figure 4.2 (page 55) I present the criticality results on merge sort for both Spearman's footrule error and max displacement error. The top graph shows the Spearman's footrule error (also known as positional error) with and without a failure on the given operation at a 10% background failure rate. The middle graph shows the difference between these two lines at the 10% background failure rate, as well as the criticalities obtained at a 0% and 20% background failure rates. The final graph shows max displacement error.

In the two bottom graphs we see a structure that cannot be found for strict correctness error and which involves, roughly, four spikes in criticality at 0, 40, 100, and 140 comparisons. We also see, towards the end of the program, a tail of about 50 operations (from about 180 to about 230) that all have low, but still non-zero, criticalities. We also see that max displacement error behaves very similarly to Spearman's footrule error at a 0% background failure rate, but is very muted when the background failure rate goes above 0%. These structures are discussed in Section 7.2.2.

In Figure 4.3 (page 56) I present the results for max inversions and Spearman's footrule error measures on quicksort. Max inversions at a 0% error rate presents a constant criticality across all comparison of ≈ 0.02. This is equivalent to a single item being a single spot out of place. Whenever there are background failures this criticality measure falls to a much lower level, however later operations have higher criticalities than earlier operations. Note that Spearman's footrule error obtains a large spike before the first n comparisons, followed by an otherwise high value for the first n comparisons and what appears to be a secondary spike before the next n comparisons followed by criticalities that trail off towards zero at the end of the algorithm.

Finally, in Figure 4.4 (page 57) I present the criticality results for full bubble sort on Spearman's footrule error. In this algorithm, no criticality is measured on any comparison except the last n comparisons, at 0% background failure rate. However, as the background failure rate is turned up, we see a periodic pattern of increased criticalities on the middle operations of every pass through the algorithm.

In this chapter, I only conveyed my immediate impressions concerning the experimental results on sorting. In Chapter 7 I will explore the consequences of these data in greater detail and speculate on their meaning.

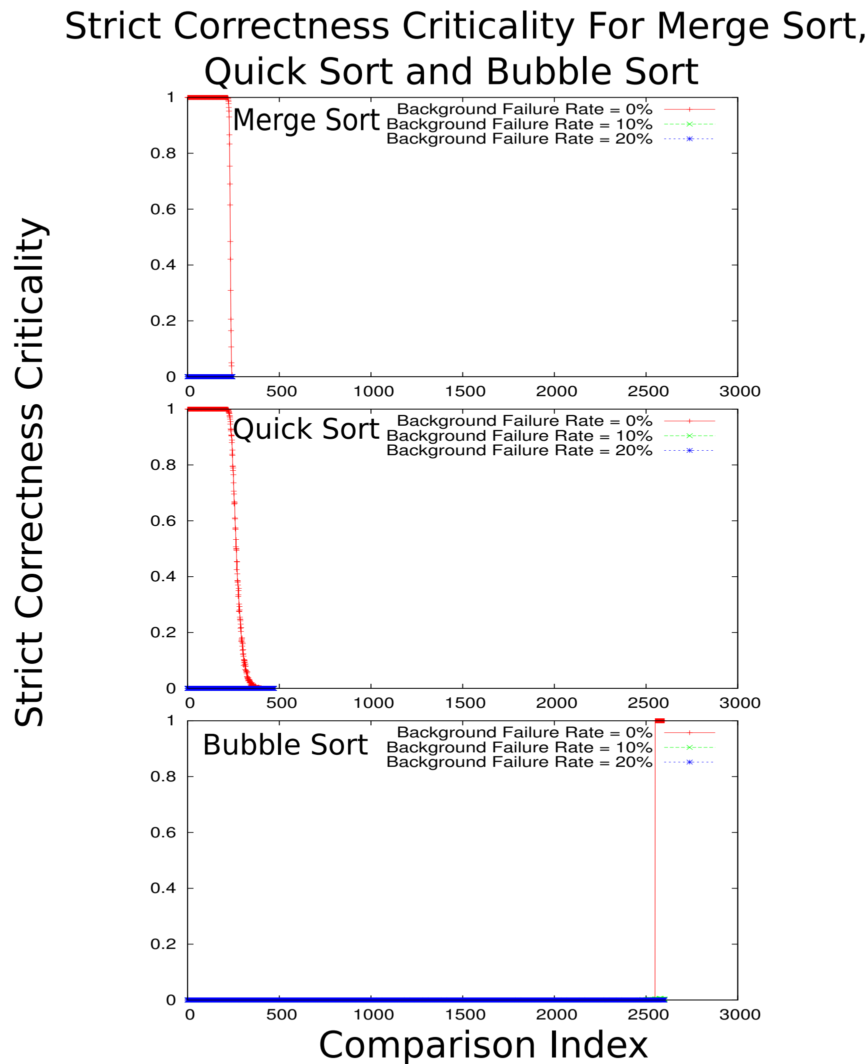


Figure 4.1: **Strict Correctness Criticality on Sorting Algorithms** Extremal values dominate in a plot of strict correctness criticality (y axis) vs. the comparisons executed during a sort ('Comparison Index'; x axis): Most faults are either critical or not critical. See Section 4.4 for details. Figure reprinted from [2] with permission.

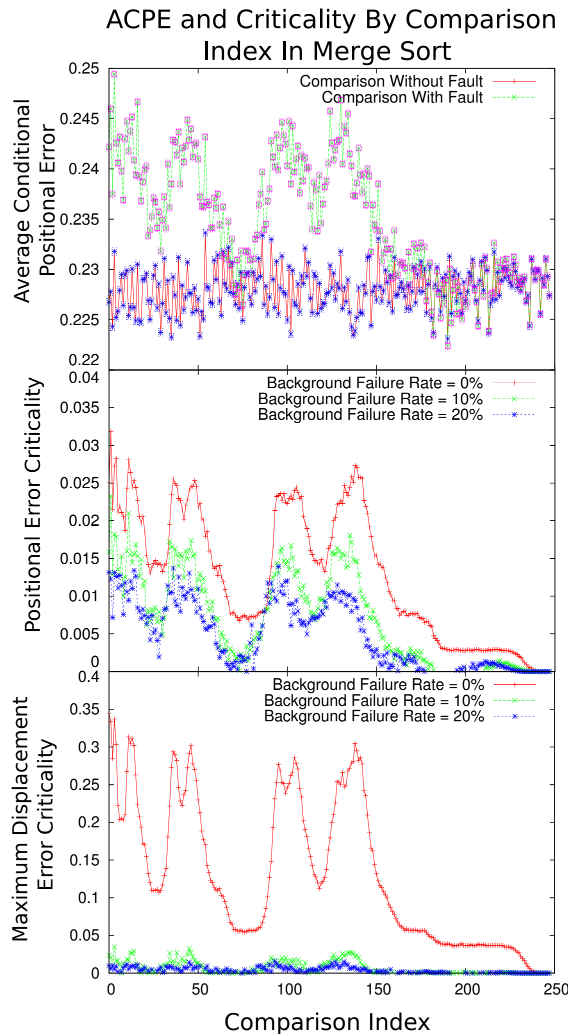


Figure 4.2: **Quantified Correctness on Merge Sort** The average conditional positional error curves (*top* graph), corresponding to the estimated error with and without the fault at the given comparison index, and the positional error criticality (*middle* graph), both based on a 10% background error rate. Note that the purple and blue boxes are error bars. See text for details. Figure reprinted from [2] with permission.

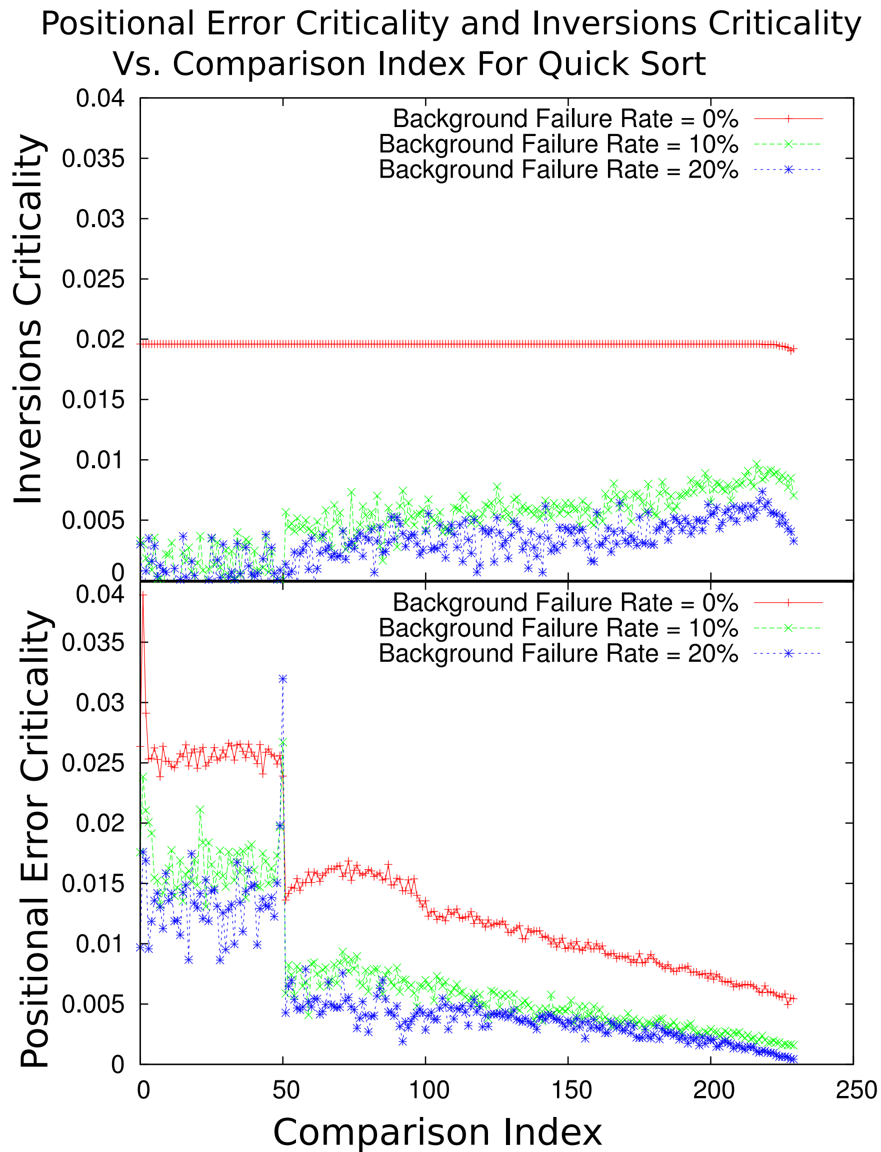


Figure 4.3: **Quantified Correctness Results on Quicksort** In quicksort we see that the choice of error measure can effect which comparisons are seen as most critical. With a positional error measure the first N comparisons are the *most* critical. However, under the inversions error measure the first N comparisons are the *least* critical comparisons. See text for details. Figure reprinted from [2] with permission.

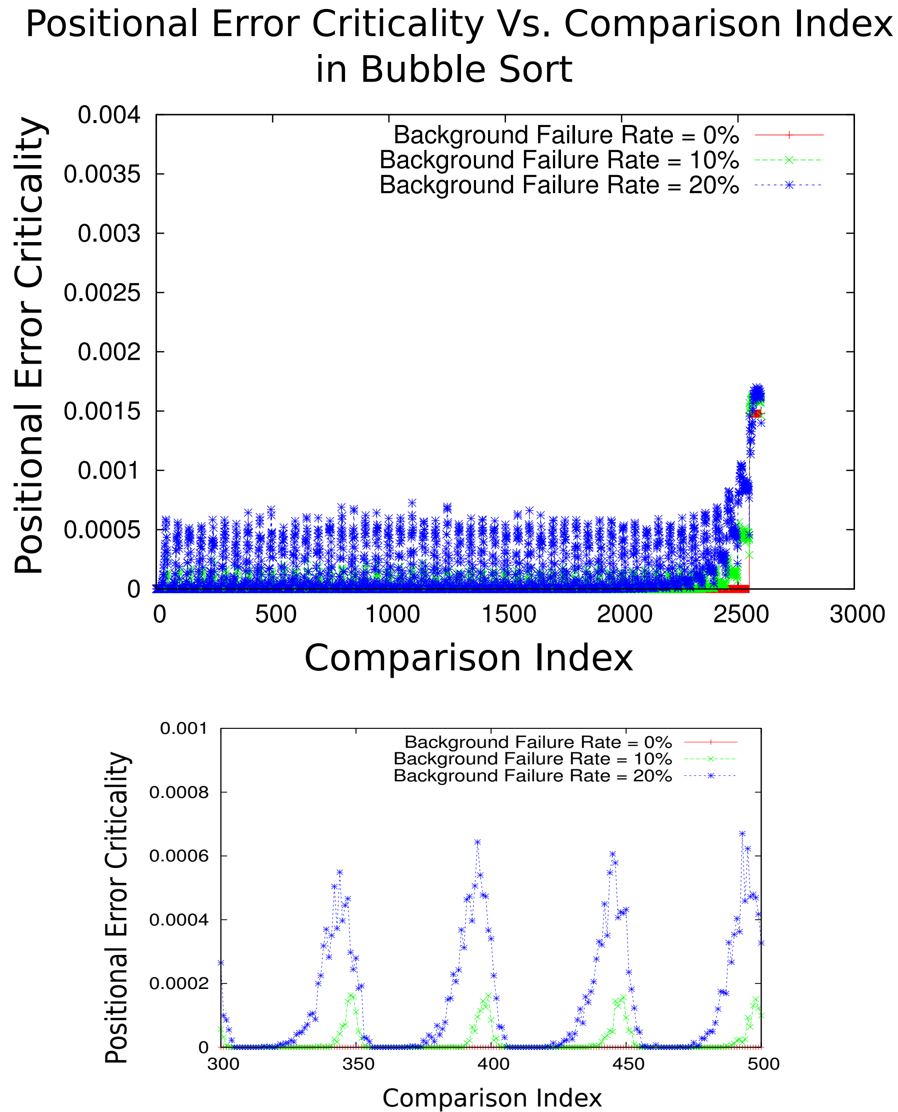


Figure 4.4: **Periodic Criticality in Bubble Sort** When facing a non-zero background error rate bubble sort presents periodic criticality behavior. The bottom graph is a subgraph of the top graph. See text for details. Reprinted from [2] with permission.

Chapter 5

Criticality and Armoring in Matrix Multiplication

5.1 Author Contribution Statement

I am the lead author of the paper underlying this chapter under the supervision of Dave Ackley (Associate Professor Emeritus, UNM Computer Science). I outlined, developed, and wrote the experiments that provided the data for this chapter, including *Criticality Explorer*. I also outlined, developed, and wrote the contents of this chapter, developed appropriate figures, and formatted the chapter. Dr. Ackley aided in the editing of this chapter and with the development of figures for this chapter.

5.2 Publication Notes

Citation: T. B. Jones and D. H. Ackley, "Damage reduction via white-box failure shaping" in *International Symposium on Search Based Software Engineering*, pp. 213-228, Springer, 2018.

Editors: Thelma Elita Colanzi Lopes and Phil McMinn

Received: March 29, 2018

Accepted: June 6, 2018

Published: September 8, 2018

Acknowledgment: Adapted by permission from Springer Nature Customer Service Centre GmbH: Springer Nature *International Symposium on Search Based Software Engineering* "Damage Reduction via White-Box Failure Shaping," T. B. Jones and D. H. Ackley, © 2018

Formatting: Substantial parts of this chapter are derived directly from the original text, however some parts have been edited for clarity or modified to fit with this dissertation. Figures are mostly unchanged, though they have been modified to fit with this dissertation.

Data Availability Statement: Data is available at
`https://github.com/ThomasBJones2/Robustness_Dissertation_Data`.

Funding: No funding was provided for this work.

Competing Interests: The authors have declared that no competing interests exist.

Supporting Information: Supporting information for this chapter can be found in Chapter 3.

5.3 Introduction

The second study I present concerns the criticality of `add`, boolean `check`, and scalar `multiply` operations in multiplication algorithms. In this study:

- I examine the criticality behavior of two scalar multiplication algorithms and two matrix multiplication algorithms.
- The program input for the scalar multiplication algorithms are integers represented as bit strings generated by flipping bits using a binomial distribution. For the matrix multiplication algorithms the inputs are are square matrices of integers of 10 bits with dimensions of size 2^l for some l.
- For the scalar multiplication algorithms I use absolute difference, log absolute difference, and absolute percentage difference error measures.
- I consider only failures on `add`, `check`, and scalar `multiplication` operations executed by each program.
- The failure mode of the `add` and scalar `multiply` operations randomly flip one bit in their outputs. The failure mode of a boolean `check` returns the *opposite result* as the 'correct' result of the boolean `check` operation.

5.4 Failure Shaping on Scalar Multiplication

I used *Criticality Explorer* to measure failure point criticalities in the $O(N^2)$ naive scalar multiplication and $O(N^{lg_2 3})$ Karatsuba multiplication algorithms [96]. I note that while Karatsuba, outlined in Algorithm 1 has a faster asymptotic runtime than naive multiplication, its runtime is actually longer on the input sizes I tested.

I employed the same input generator for both algorithms, producing random N bit vectors interpreted as two's-complement integers. I applied three different error measures to each algorithm: *absolute value*, *absolute logarithmic value*, and *absolute percent value*. Given the correct answer C and an incorrect output I, *absolute value* returned $|C - I|$, *absolute logarithm value* returned $ln(|C - I| + 1)$, and *absolute percent value* returned $\frac{|C-I|}{|C|+1}$.

Algorithm 1: Karatsuba Multiplication

```
1:  procedure KARATSUBA(x, y)
2:      if bl(x) ≤ 8 then                          ▷ bl(x) is bit length of x
3:          return Naive(x,y)
4:      end if
5:      if x ≡ 0 or y ≡ 0 then
6:          return 0
7:      end if
8:      x1 2^m + x0 = x                            ▷ m ≤ min(bl(x),bl(y))/2
9:      y1 2^m + y0 = y
10:     z1 = Karatsuba(x1, y1)
11:     z2 = Karatsuba(x0, y0)
12:     z3 = Karatsuba(x1 + x0, y1 + y0)
13:     k = z3 − z2 − z1
14:     return 2^{2m} z1 + 2^m k + z3
15: end procedure
```

I defined failure interfaces for `check` and `add` methods used by both algorithms. The Boolean `check` method returns $true$ if the given bit of a number is 1 and $false$ if the number is 0. Its failure method returns the opposite result when a failure occurs. The `add` method returns the sum of two numbers and its failure method randomly flips one bit on the output when called. Since Karatsuba called naive multiply at small input sizes, both algorithms would call a `common addition` operation while Karatsuba called a special `Karatsuba add`, located only in the Karatsuba recursion function, separately.

5.4.1 Criticality Assessment Results on Scalar Multiplication

Figure 5.1 (page 68) presents example criticality distributions and sample sizes for both naive multiplication and Karatsuba multiplication algorithms. I obtained similar results with 100, 200, and 500 bit numbers; here I focus on $N = 100$ and $N = 500$. Naive multiplication's `check` interface follows a log-linear growth pattern at all input sizes. Karatsuba's `check` and `add` interfaces display repeating three maxima criticality patterns at both scales, in keeping with its triple recursive call.

Both distributions maintain tight bounds on the log-linear scale for all but the final few failure points. As sample size decreases, standard error increases. I chose not to display *absolute value* and *absolute percent value* because both graphs appear flat on all but one or two failure points. For scalar multiplication, error is grows exponentially. Figure 5.1 (page 68) also illustrates the median criticality for each failure interface. For the `add` interfaces, naive multiply's medians lie below Karatsuba's. With `check` the inverse is true.

5.4.2 Failure Shaping Results on Scalar Multiplication

In Figures 5.8 (page 75), 5.9 (page 76), and 5.10 (page 77) I show failure shaped error results against i.i.d. failure model outcomes for each failure interface and error measure at input size 100. My sample size was 1000 on both algorithms and all three error measures. Tested error rates, ϵ, were evenly distributed in $[0, 0.1]$.

Failure shaping, as I have outlined it, can distort the underlying failure rate, ϵ. I believe this is caused by inaccuracies in the measured median failure rate, or by changes in the run time caused when ϵ increases from zero. Therefore, I report results at the *actual observed rate of failure*, ϵ', in my graphs. Nonetheless, ϵ' is generally within 5% of ϵ.

Failure shaping cuts the average absolute logarithmic failure by about 30. It cuts the absolute value and absolute percent value by between 10% and 90%. Further, the Karatsuba algorithm benefits more from failure shaping than naive multiply.

5.5 Failure Shaping Matrix Multiplication

To show how leverage works on multiple *algorithmic* scales—across hierarchies of operations within algorithms—I used *Criticality Explorer* to perform an economization procedure on two matrix multiplication algorithms. The first is the $O(N^3)$ naive matrix multiplication algorithm. The second algorithm is Strassen's algorithm [97]. This algorithm runs in $O(N^{2.8})$ and is the first divide and conquer matrix multiplication algorithm found to run faster than $O(N^3)$. It is outlined in Algorithm 2 (page 78) .

The input generators used for both algorithms were randomly generated matrices of size $N \times N$ with 10 bit integers for each element.

I employed three error measures: the *Frobenius norm* (FN, also known as the matrix euclidean distance), the *infinity norm* (IN), and the *logarithmic Frobenius norm* (LFN) given in Equations 5.1, 5.2, and 5.3 respectively. In these equations, M_c and M_i are the correct and incorrect matrices and e_i and e_c are the correct and incorrect matrix elements at the same position in either matrix.

$$FN(M_i, M_c) = \sqrt{\Sigma_{\forall e_c, e_i \in M_c, M_i} (e_c - e_i)^2} \tag{5.1}$$

$$IN(M_i, M_c) = argmax_{\forall e_c, e_i \in M_c, M_i} |e_c - e_i| \tag{5.2}$$

$$LFN(M_i, M_c) = log(FN(M_i, M_c) + 1) \tag{5.3}$$

I reused the scalar multiplication failure interfaces `check` and `add`. Both naive and Strassen's matrix multiply used the `naive scalar multiplication` sub-algorithm to perform element-wise multiplication.

5.5.1 Criticality Assessment Results on Matrix Multiplication

Figure 5.11 (page 79) presents example criticalities for both algorithms on selected operations and scales. I note here that the first major structure I observe in the failure shapes of the two matrix multiplication algorithms lies in the differences between them. Naive matrix multiplication has, in general, a lower criticality. Its criticalities are also flatter than those found in the highly structured Strassen's algorithm. At every test scale we see spikes in both `add` and `check` operations about 2/7, 3/7 and 5/7 through the algorithm run on both the infinity and Frobenius norm error measures for Strassen's algorithm.

This structure can also be seen on the *log Frobenius norm* error measure. However, it becomes harder to disentangle at larger scales as the data is squeezed. To see criticality structures on this error measure at large scales, we focus on a smaller set of failure points. Figure 5.21 (page 89) shows a view of the *log Frobenius norm* criticalities from failure point 13000 to 18000 on input size 16. The structure we see is analogous to those seen from failure points 275 to 350 at input scale 4, and from 2000 to 2500 at input scale 8.

As with scalar multiplication, criticality standard error increases with decreasing sample size at the end of both algorithm's failure shape.

In naive multiply, criticalities lie close to both the average and the median criticality. By contrast, in Strassen's algorithm important operations are outliers with criticalities often an order of magnitude greater than the median.

5.5.2 Failure Shaping Results on Matrix Multiplication

Figures 5.25 (page 93), 5.26 (page 94), and 5.27 (page 95) show the direct failure shaping results on naive and Strassen's matrix multiply algorithms plotted against an i.i.d. failure model. 1000 samples were taken at each percentile in $[0, 0.1]$. We can see that direct failure shaping produces roughly 40% error reductions compared to a baseline i.i.d. error model.

Monte Carlo sampling is a powerful statistics-gathering method, but its simulation costs here grow with the number of failure points in the system under test. *Criticality Explorer* can be connected to the AWS Lambda on-demand compute service [98], allowing investigators to trade money for time by performing parallel assessments in the cloud. As an example, the data presented in this paper was produced for under $320 in cloud costs—with the majority of that consumed by the scale 16 criticality assessments.

Even assuming such a large-scale infrastructure, though, brute force Monte Carlo costs will become prohibitive as the software stack under test grows deeper, placing more computational levels between the hardware and the end-user error measures. In the next section I introduce 'proxy criticalities'—an approach to evaluate such multilevel software that not only slashed assessment costs, but also, I found, even improved performance.

5.6 Proxy Failure Shaping

I took advantage of method level failure interface flexibility to speed up the failure shaping procedure by using *proxy criticalities*. Rather than measuring each failure point's criticality, I instead measured the criticality of a *proxy method*—a method that stands in for those methods originally intended to fail.

In this section experiments continued to make use of fallible `check` and `add` operations. However, each `add` and `check` failure interface was only called as part of a `scalar multiplication` method. I wrote a failure interface that randomly flipped one bit in a `scalar multiplication` invocation's output. Thus, criticality assessment costs on matrices with element size e require $\sim 1/e$ resources using `scalar multiplication` as a proxy for `check` and `add`. For example, the size 32 proxy algorithm assessment cost less than $14 on AWS lambda.

Using this failure interface I measured each multiplication failure point's criticality. Figures 5.22 (page 90), 5.23 (page 91), and 5.24 (page 92) show criticality assessment results on scalar multiply operations employed by both matrix multiplication algorithms. These criticalities and their median value were then employed to make decisions about the reliability budgeting of every `check` and `add` failure point that occurred *during* each multiplication operation's execution.

5.6.1 Criticality and Failure Shaping Results on Scalar Multiplication Proxy Method

Figures 5.22 (page 90), 5.23 (page 91), and 5.24 (page 92) present example proxy operation criticality distributions for both naive and Strassen's matrix multiply at input sizes 4, 8, and 16 respectively. Assessing criticality on `scalar multiplication` cut run times by an order of magnitude. Figure 5.28 (page 96) shows proxy failure shaping results on size 32 matrices.

As with criticality measurements on the `check` and `add` operations, `multiply` operations are flat for naive matrix multiply and structured for Strassen's multiply. `Scalar multiply` failure shapes in Strassen's matrix multiplication algorithm also have similar distributions to `check` and `add` operations. I can still find criticality spikes roughly 2/7, 3/7 and 5/7 of the way through the algorithm.

Figures 5.25 (page 93), 5.26 (page 94), and 5.27 (page 95) show proxy failure shaping results on `check` and `add` failure interfaces using the `scalar multiply` failure interface as a proxy. These results are compared to the baseline i.i.d. model results, and the results from the simple failure shaping procedure applied in Section 5.5.2. As can be seen, proxy failure shaping can work as well as direct failure shaping.

In this chapter, I only conveyed my immediate impressions concerning the experimental results on matrix and scalar multiplication. In Section 7.2.2 I will explore the consequences of these data in greater detail and speculate on their meaning.

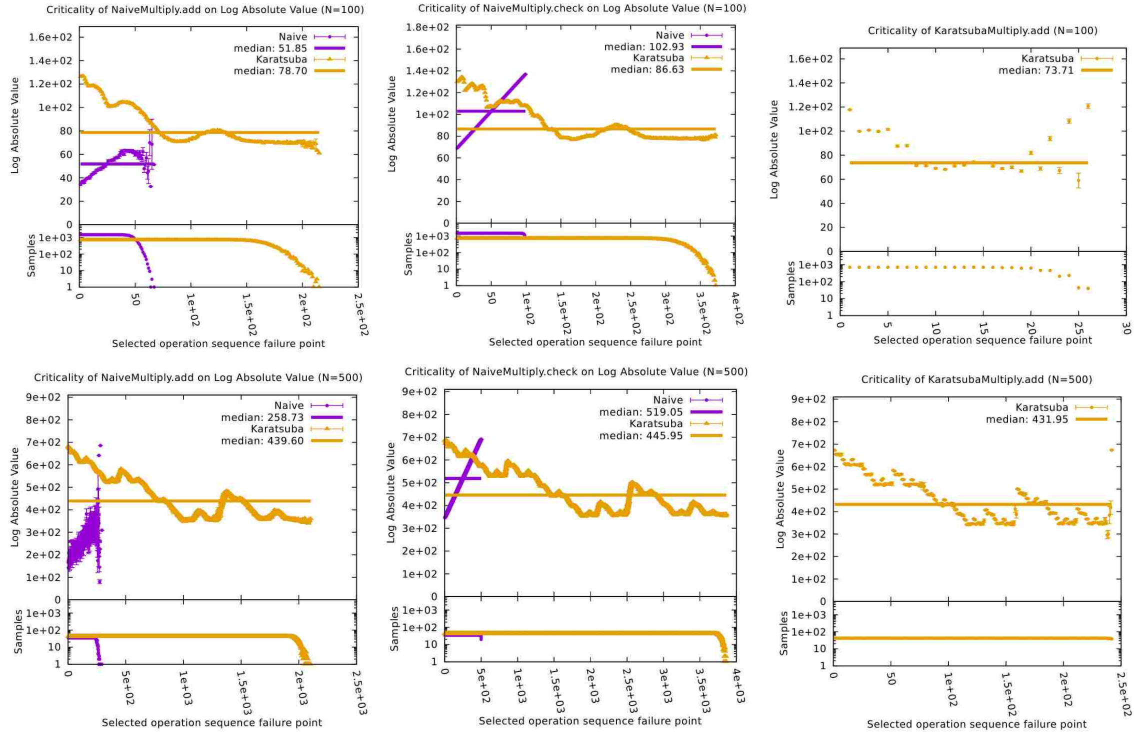


Figure 5.1: **Scalar Multiplication Criticality Assessment Results.** *Absolute log* criticality assessment results for both the naive and Karatsuba multiplication algorithms. The left graphs show results on the `common add` failure interface, the middle graphs show results for the `check` failure interface, and the right graphs show results for the `Karatsuba add` interface, which only the Karatsuba algorithm called. The top graphs are for input size $N = 100$ while the bottom graphs are for size input $N = 500$. The naive multiplication algorithm shows a log-linear growth pattern in criticality on the `check` operation and a log-linear growth pattern on half of the `add` operation while the Karatsuba algorithm shows a group of three maxima on every interface. Note that graphs have different x and y axes. See text for details and Figures 5.2 (page 69), 5.3 (page 70), 5.4 (page 71), 5.5 (page 72), 5.6 (page 73), and 5.7 (page 74) for expanded views of each graph. Reprinted from [1] with permission.

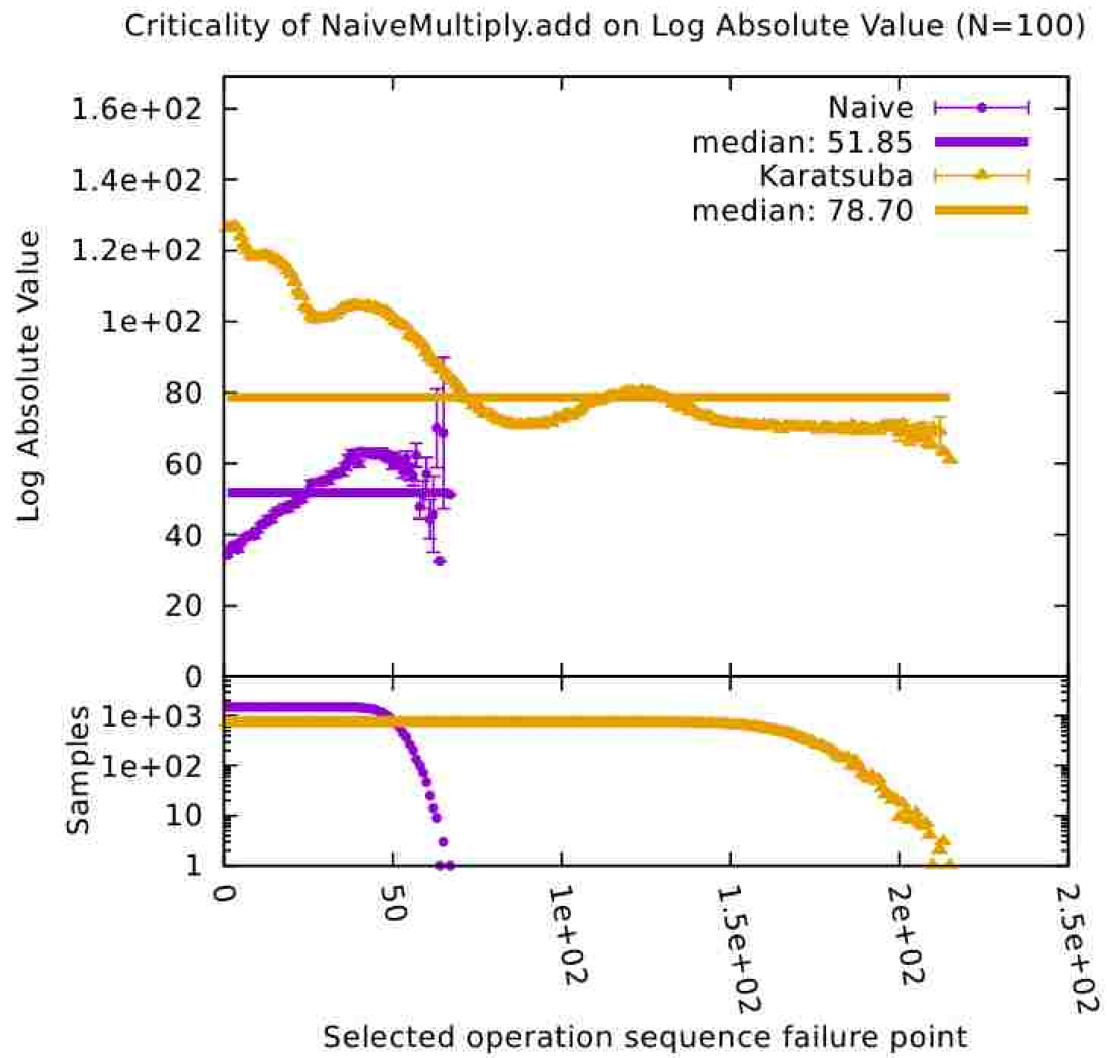


Figure 5.2: **Expanded Scalar Multiplication Criticality Assessment Results on NaiveMultiply.add (N=100).** See Figure 5.1 (page 68) for details.

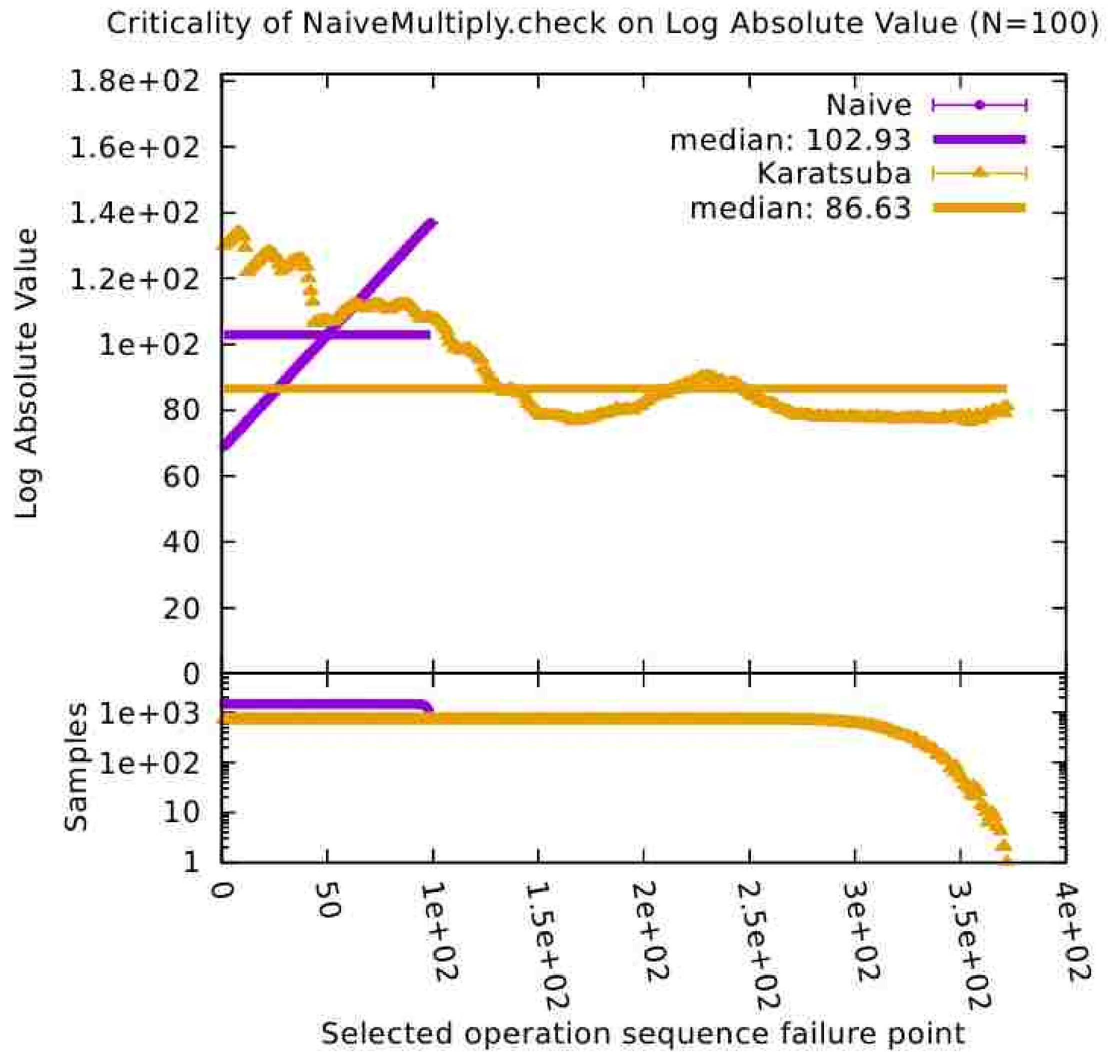


Figure 5.3: **Expanded Scalar Multiplication Criticality Assessment Results on NaiveMultiply.check (N=100).** See Figure 5.1 (page 68) for details.

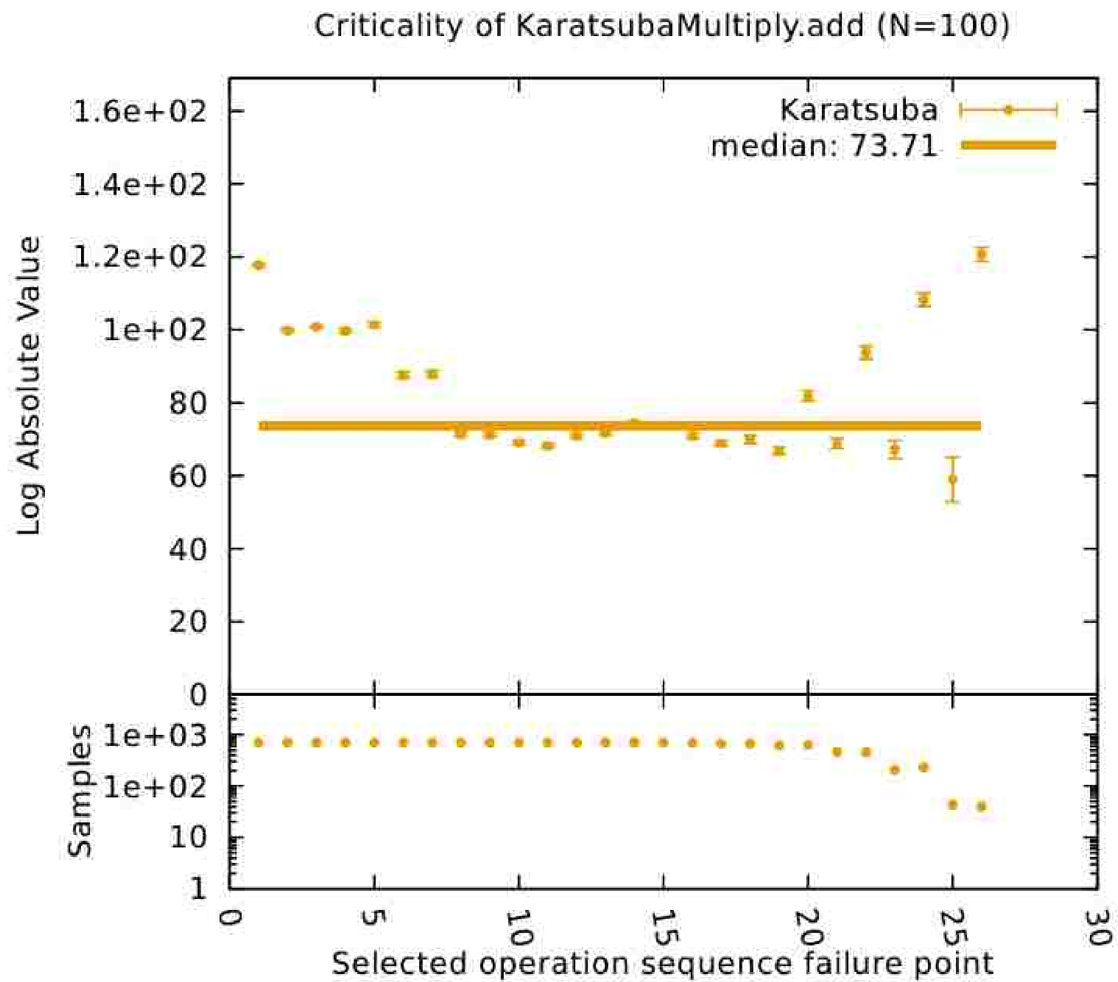


Figure 5.4: **Expanded Scalar Multiplication Criticality Assessment Results on KaratsubaMultiply.add (N=100).** See Figure 5.1 (page 68) for details.

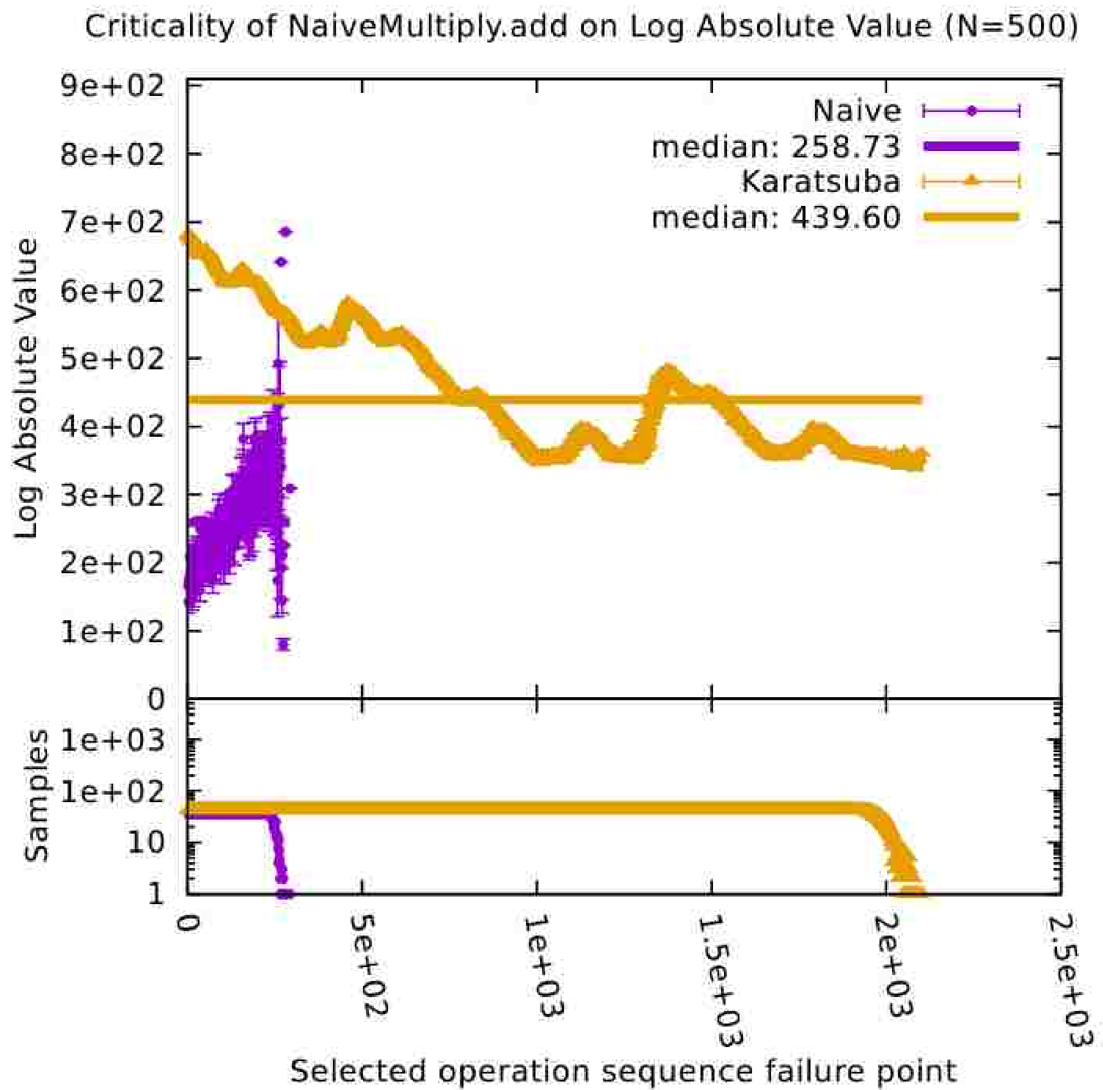


Figure 5.5: **Expanded Scalar Multiplication Criticality Assessment Results on NaiveMultiply.add (N=500).** See Figure 5.1 (page 68) for details.

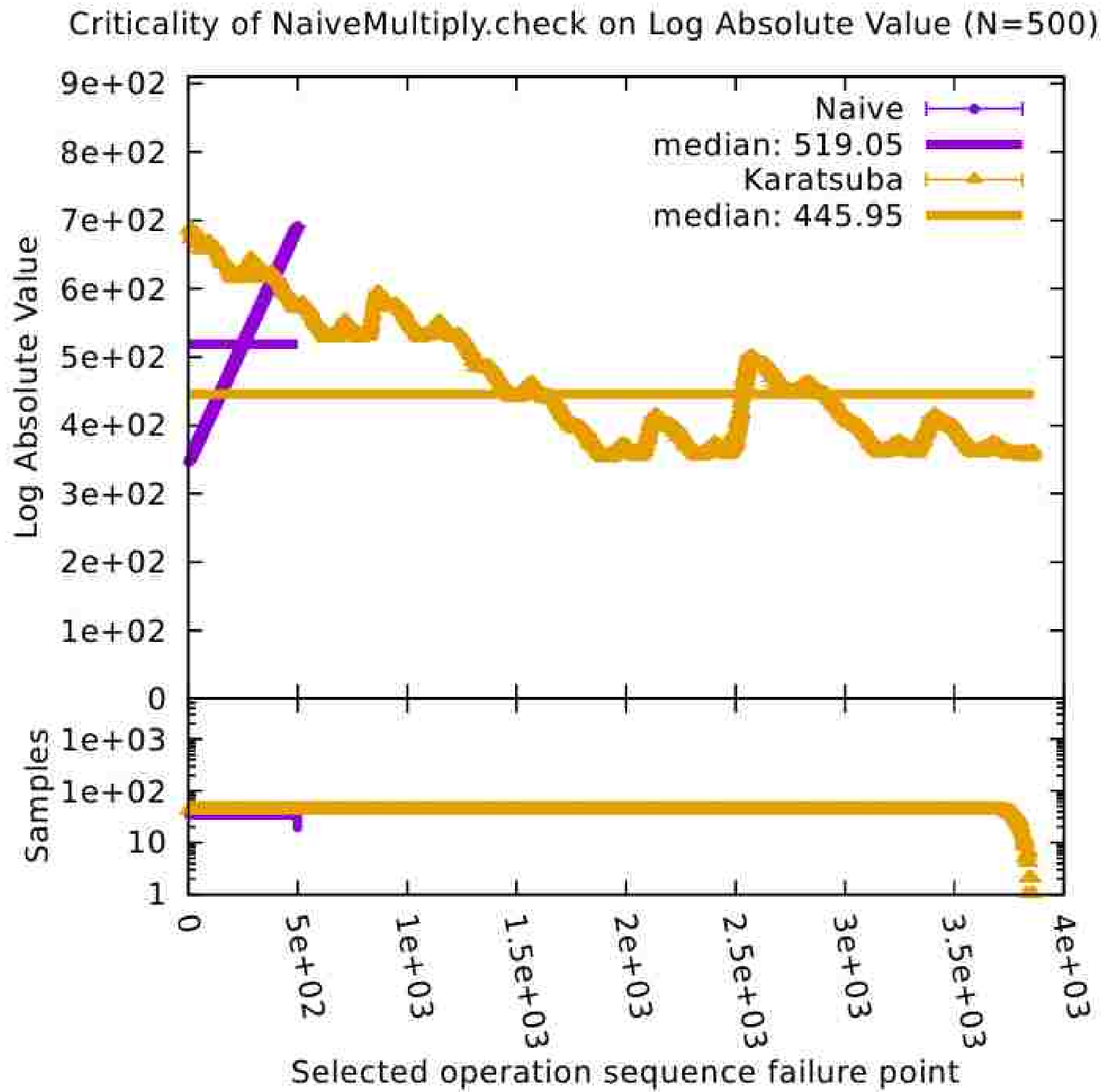


Figure 5.6: **Expanded Scalar Multiplication Criticality Assessment Results on NaiveMultiply.check (N=500).** See Figure 5.1 (page 68) for details.

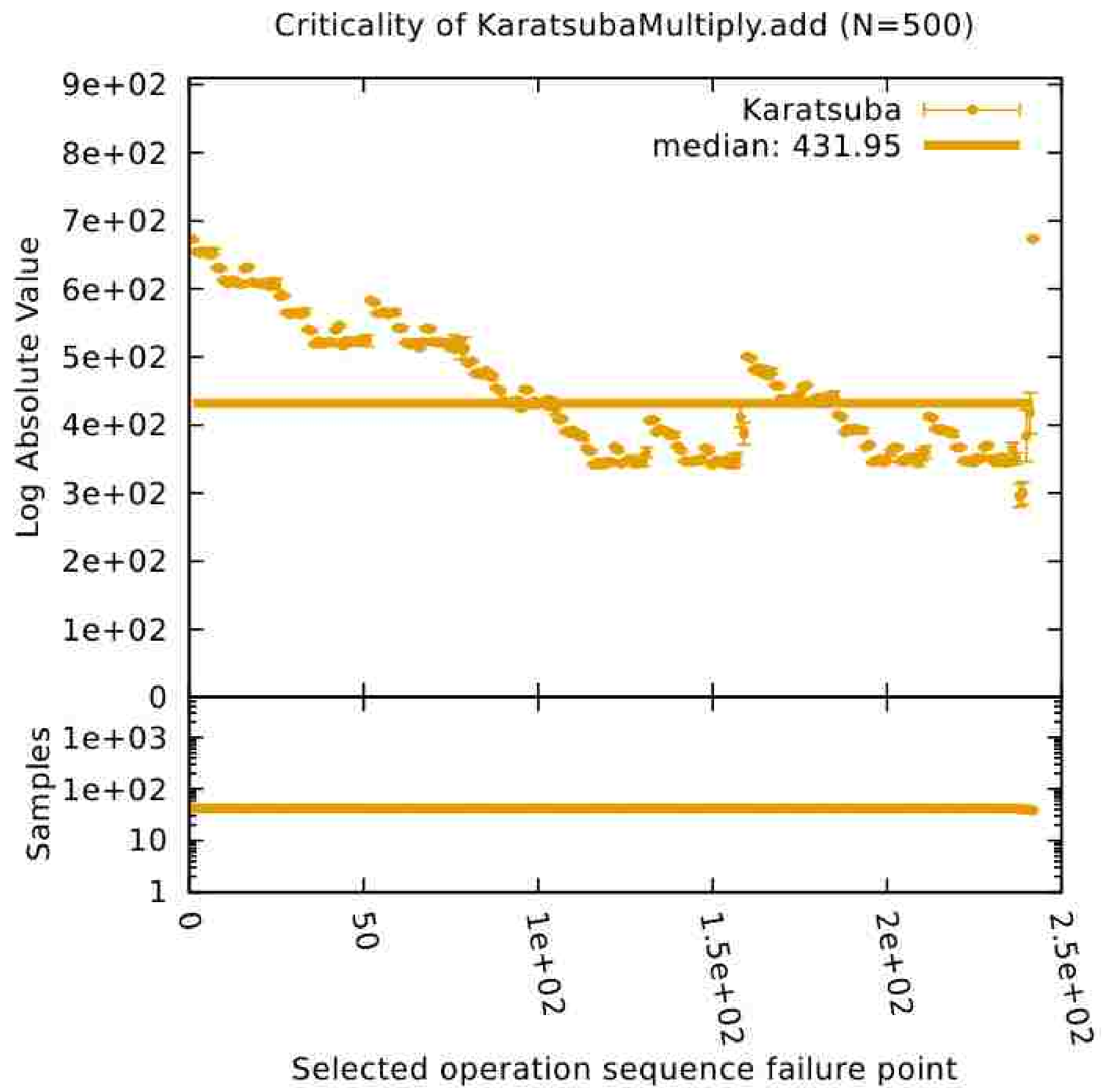


Figure 5.7: **Expanded Scalar Multiplication Criticality Assessment Results on KaratsubaMultiply.add (N=500).** See Figure 5.1 (page 68) for details.

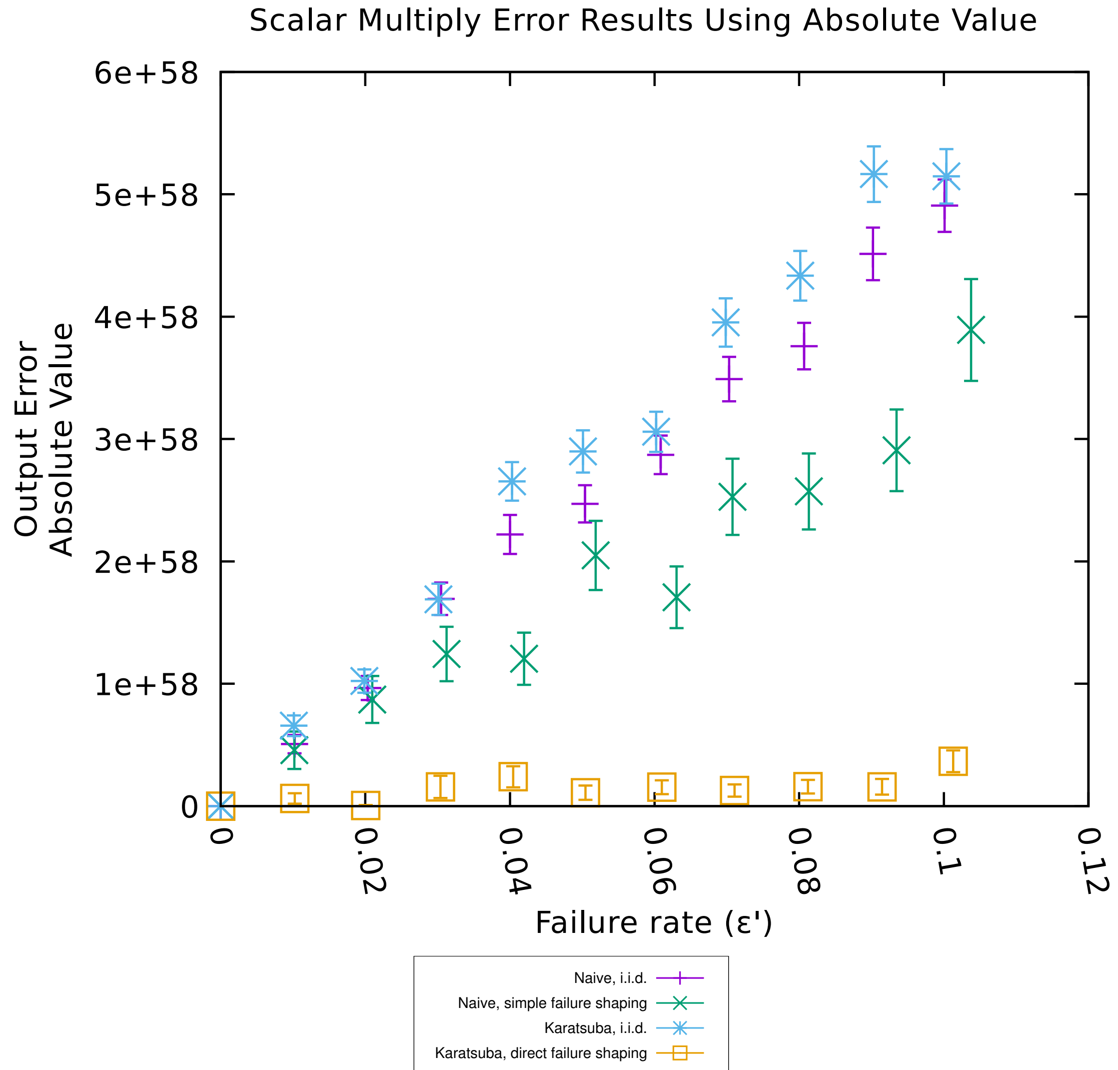


Figure 5.8: **Scalar Multiplication Failure Shaping Results on Absolute Value.** Average error rates for both the naive and Karatsuba multiplication algorithms assessed using *average absolute value* error measure on input size 100. Assessments were performed on both algorithms using a baseline i.i.d. failure model and the median value failure shaping technique. See text for details. Reprinted from [1] with permission.

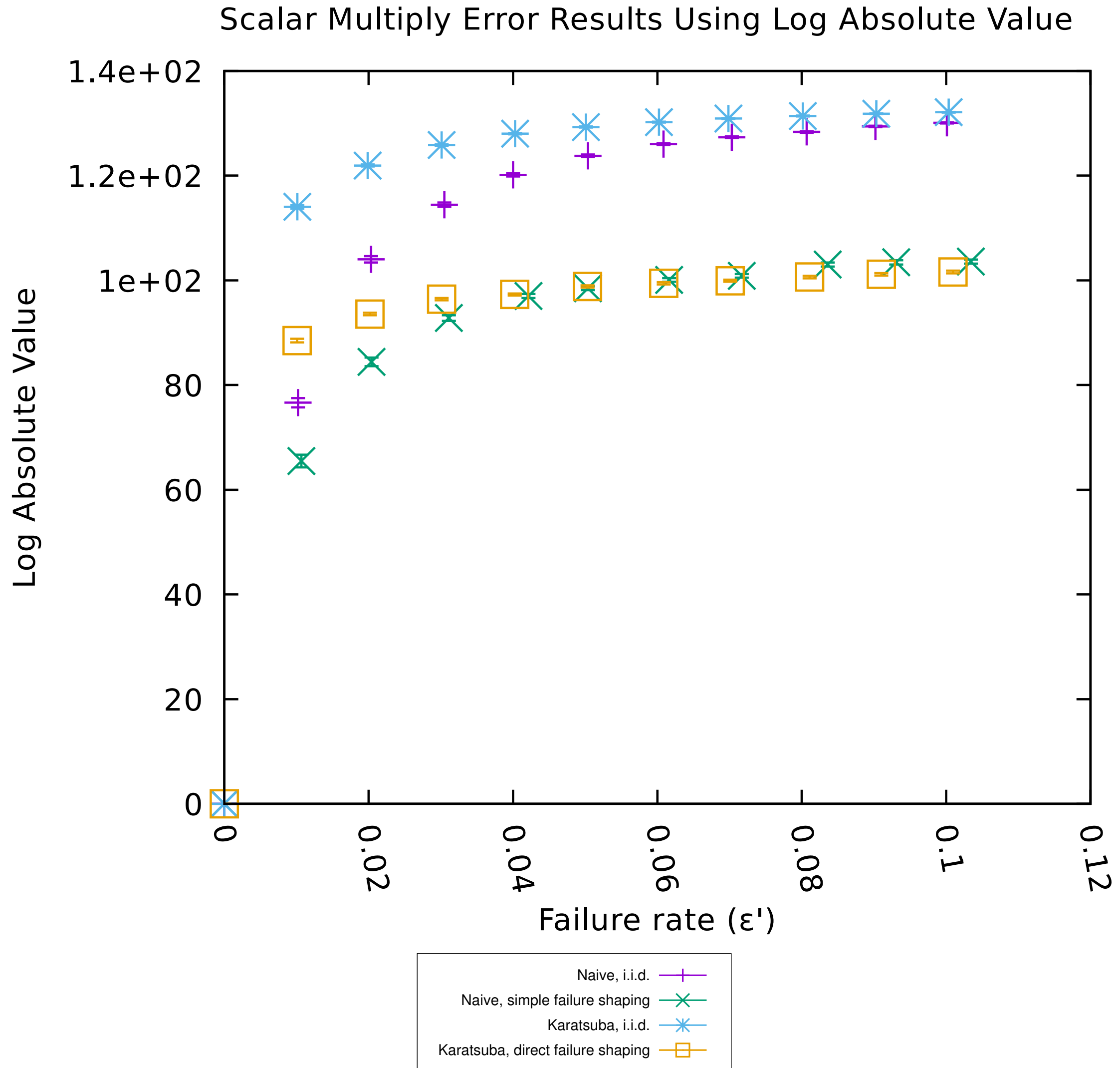


Figure 5.9: **Scalar Multiplication Failure Shaping Results on Log Absolute Value.** Average error rates for both the naive and Karatsuba multiplication algorithms assessed using *average log absolute value* error measure on input size 100. Assessments were performed on both algorithms using a baseline i.i.d. failure model and the median value failure shaping technique. See text for details. Reprinted from [1] with permission.

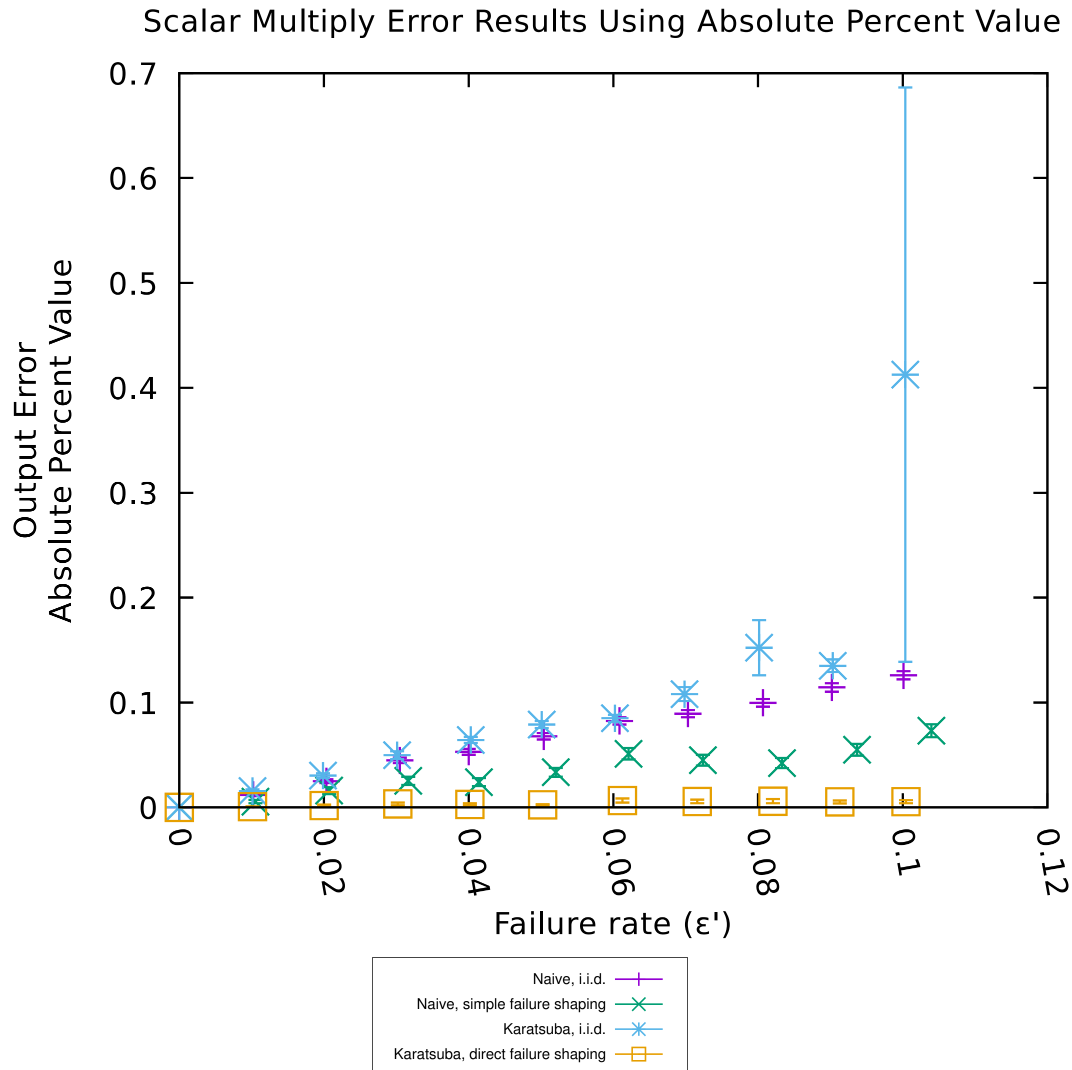


Figure 5.10: **Scalar Multiplication Failure Shaping Results on Absolute Percent Value.** Average error rates for both the naive and Karatsuba multiplication algorithms assessed using *average absolute percentage value* error measure on input size 100. Assessments were performed on both algorithms using a baseline i.i.d. failure model and the median value failure shaping technique. Percent value is represented as a number in $[0, 1]$ and is sensitive to minor changes in its denominator. See text for details. Reprinted from [1] with permission.

Algorithm 2: **Strassen's Matrix Multiplication.**

```
1: procedure STRASSEN(x, y)                    ▷ x and y are size N × N
2:     if N == 1 then
3:         return [[x[0][0]y[0][0]]]
4:     end if
5:     a = x[0 : N/2][0 : N/2]
6:     b = x[0 : N/2][N/2 + 1 : N]
7:     c = x[N/2 + 1 : N][0 : N/2]
8:     d = x[N/2 + 1 : N][N/2 + 1 : N]
9:     e = y[0 : N/2][0 : N/2]
10:    f = y[0 : N/2][N/2 + 1 : N]
11:    g = y[N/2 + 1 : N][0 : N/2]
12:    h = y[N/2 + 1 : N][N/2 + 1 : N]
13:    p1 = Strassen(a, f − h)
14:    p2 = Strassen(a + b, h)
15:    p3 = Strassen(c + d, e)
16:    p4 = Strassen(d, g − e)
17:    p5 = Strassen(a + d, e + h)
18:    p6 = Strassen(b − d, g + h)
19:    p7 = Strassen(a − c, e + f)
20:    z[0 : N/2][0 : N/2] = p5 + p4 − p2 + p6
21:    z[0 : N/2][N/2 + 1 : N] = p1 + p2
22:    z[N/2 + 1 : N][0 : N/2] = p3 + p4
23:    z[N/2 + 1 : N][N/2 + 1 : N] = p1 + p5 − p3 − p7
24:    return z
25: end procedure
```

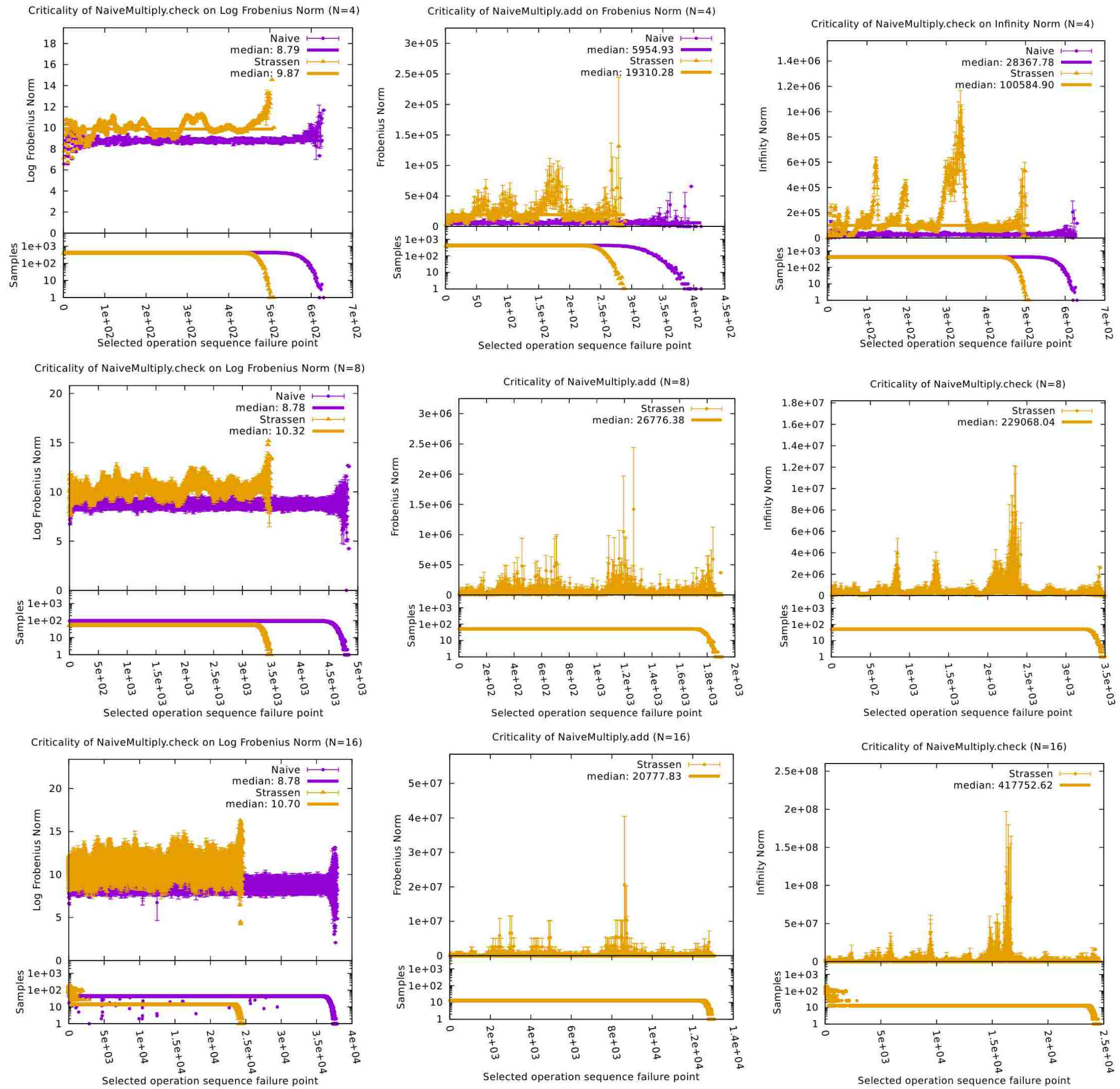


Figure 5.11: **Matrix Multiplication Criticality Assessment Results.** Criticality results for both naive and Strassen's matrix multiplication. The left graphs show *log Frobenius norm*, the middle graphs show *Frobenius norm*, and the right graphs show *infinity norm* results. The graphs on the top are for matrix input size 4, the middle set are at size 8 and the bottom set are at size 16. Naive results are omitted in the four graphs towards the bottom right to emphasize similar structures in Strassen's algorithm at multiple scales. Note that x and y axes do not match for every graph. Each error measure is valued in its own units. See text for details and Figures 5.12 (page 80), 5.13 (page 81), 5.14 (page 82), 5.15 (page 83), 5.16 (page 84), 5.17 (page 85), 5.18 (page 86), 5.19 (page 87), and 5.20 (page 88) for expanded views of each graph. Reprinted from [1] with permission.

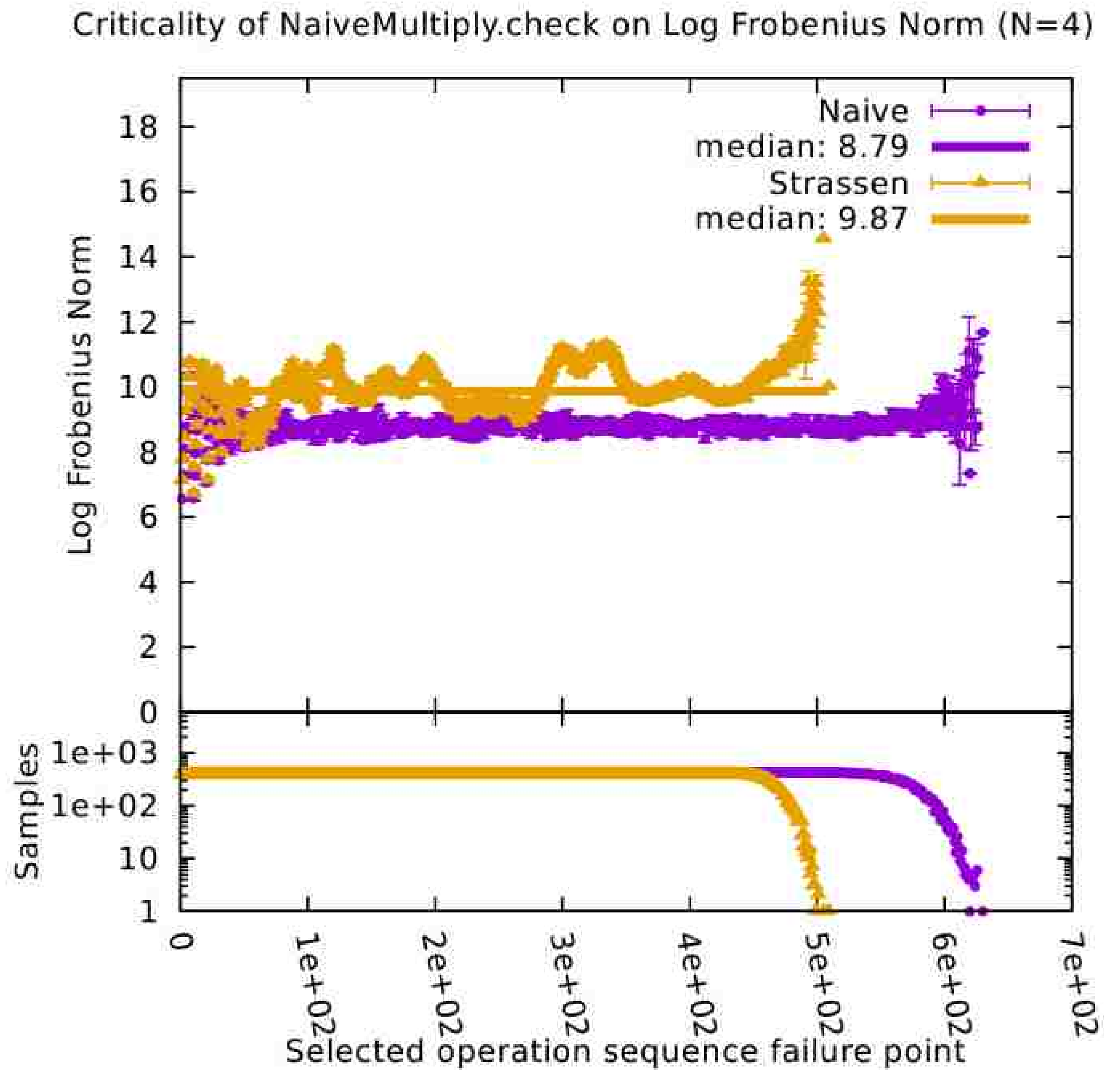


Figure 5.12: **Expanded Log Frobenius Norm Matrix Multiplication Criticality Assessment Results on NaiveMultiply.check (N=4).** See Figure 5.11 (page 79) for details.

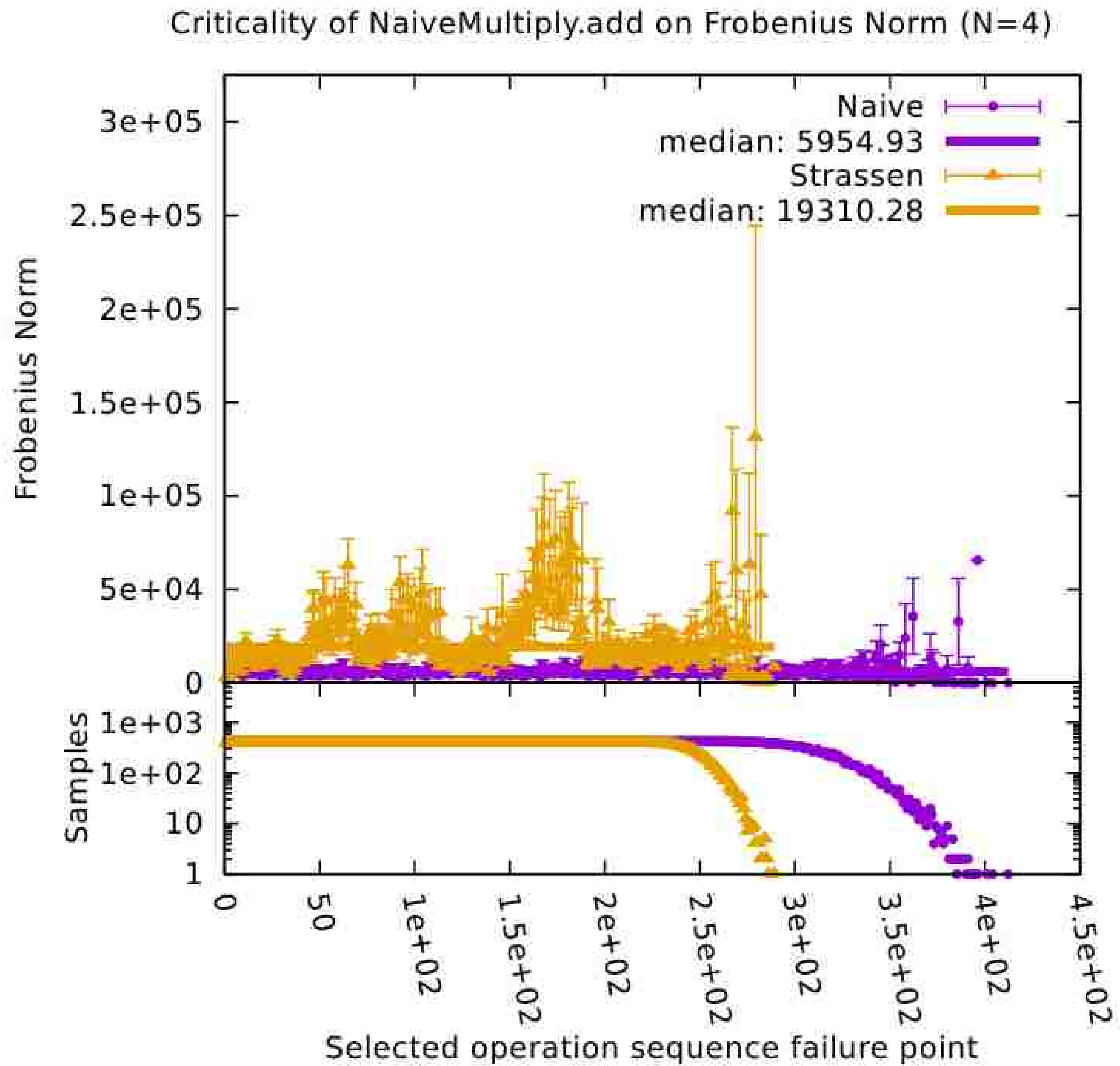


Figure 5.13: **Expanded Frobenius Norm Matrix Multiplication Criticality Assessment Results on NaiveMultiply.add (N=4).** See Figure 5.11 (page 79) for details.

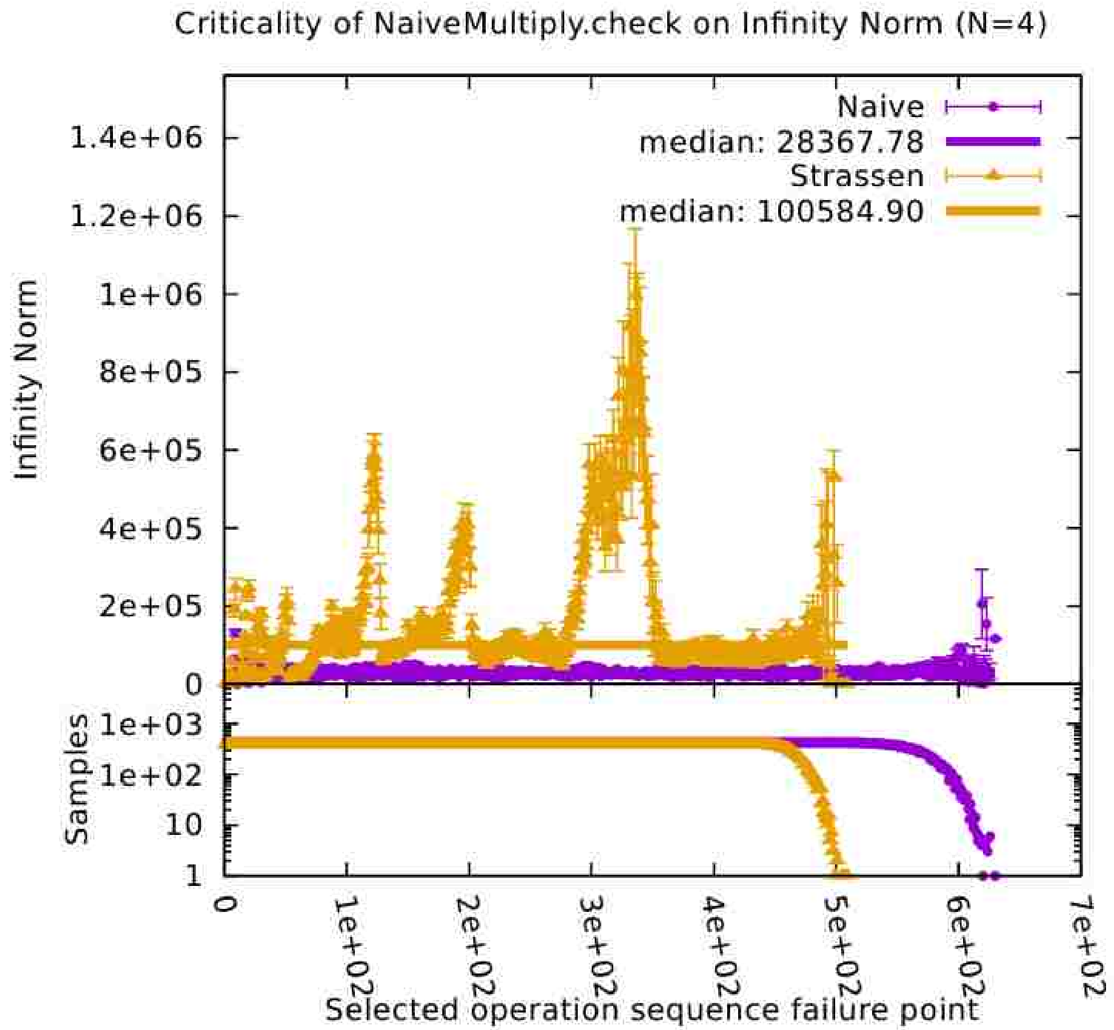


Figure 5.14: **Expanded Infinity Norm Matrix Multiplication Criticality Assessment Results on NaiveMultiply.check (N=4).** See Figure 5.11 (page 79) for details.

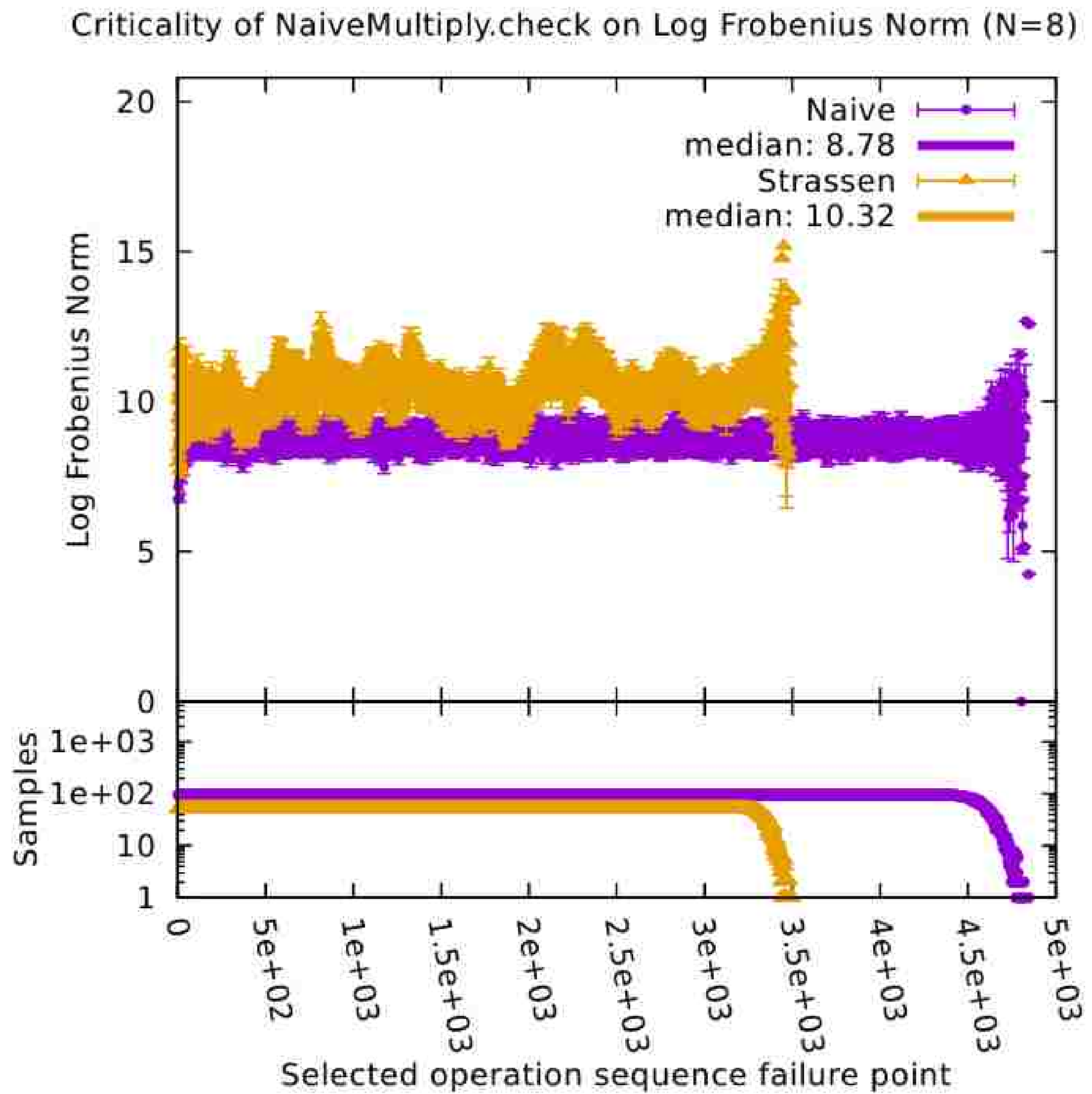


Figure 5.15: **Expanded Log Frobenius Norm Matrix Multiplication Criticality Assessment Results on NaiveMultiply.check (N=8).** See Figure 5.11 (page 79) for details.

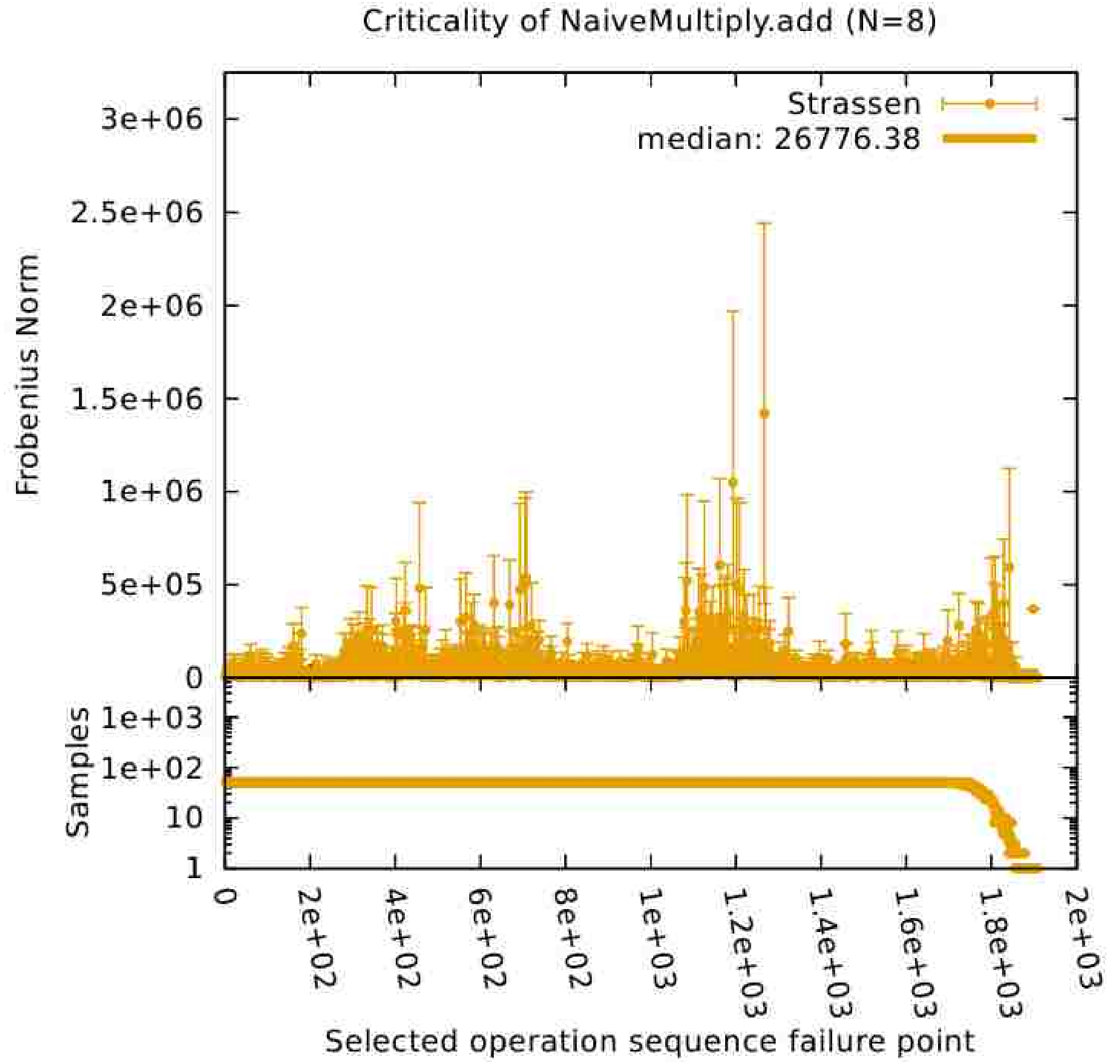


Figure 5.16: **Expanded Frobenius Norm Strassen's Criticality Assessment Results on NaiveMultiply.add (N=8).** See Figure 5.11 (page 79) for details.

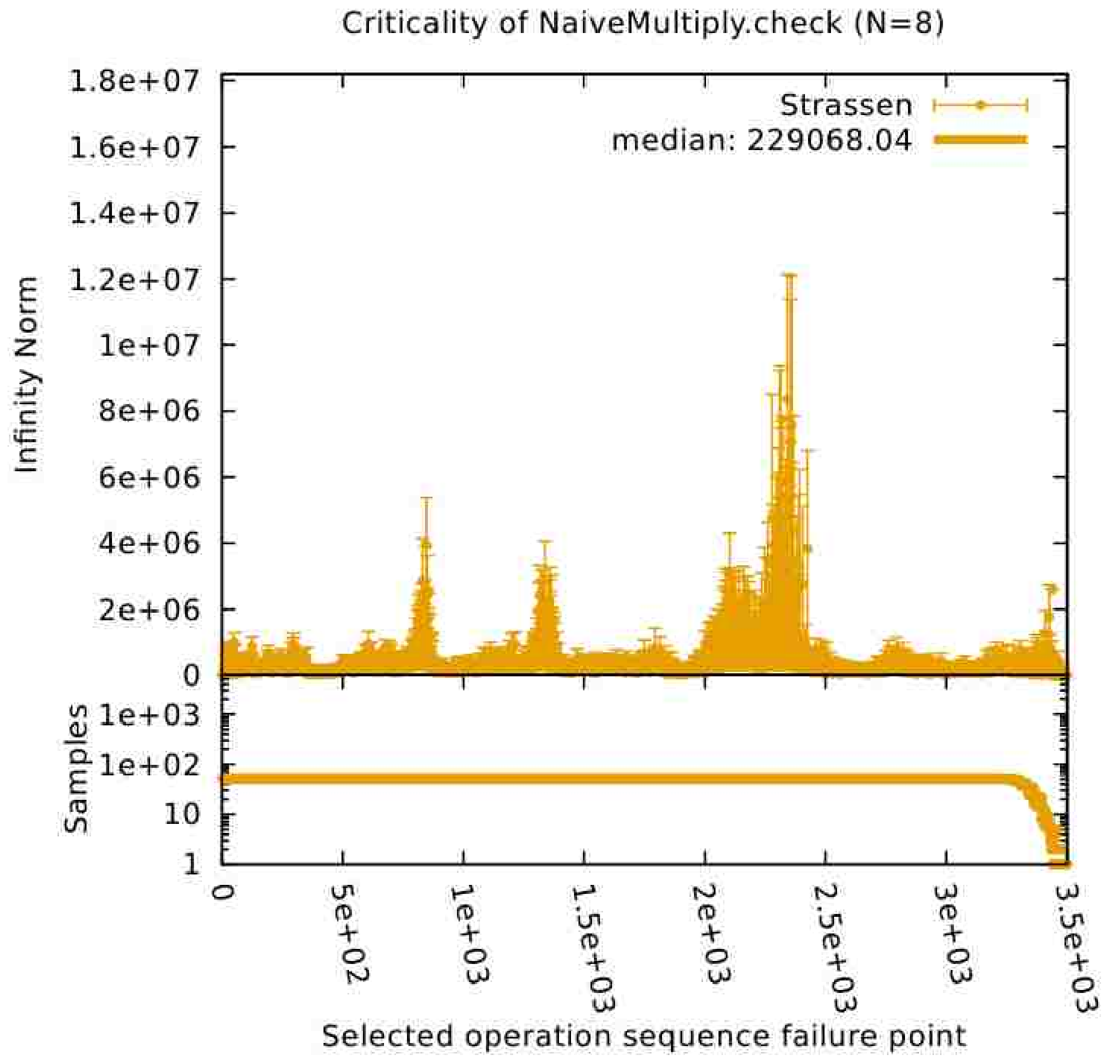


Figure 5.17: **Expanded Infinity Norm Strassen's Criticality Assessment Results on NaiveMultiply.check (N=8).** See Figure 5.11 (page 79) for details.

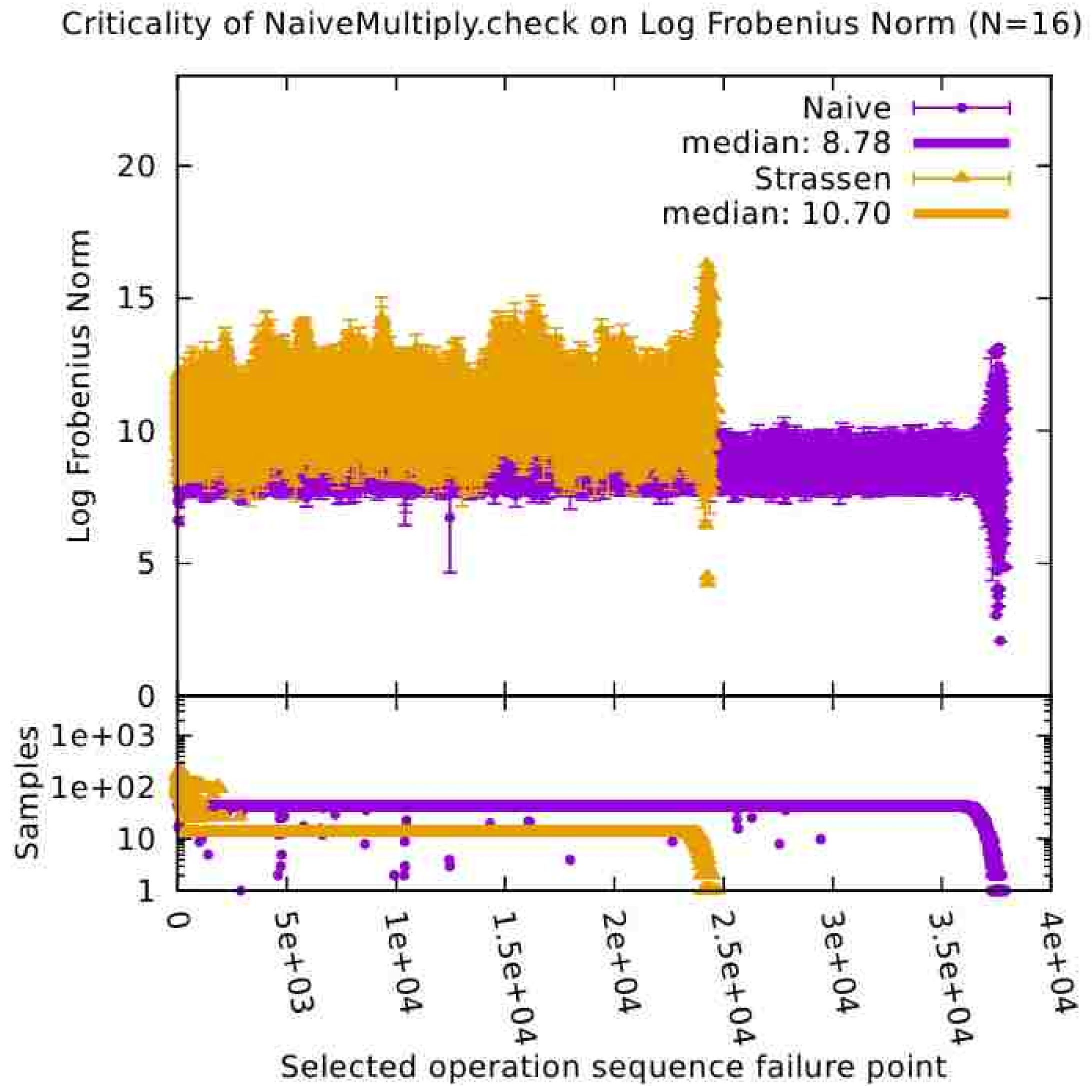


Figure 5.18: **Expanded Log Frobenius Norm Matrix Multiplication Criticality Assessment Results on NaiveMultiply.check (N=16).** See Figure 5.11 (page 79) for details.

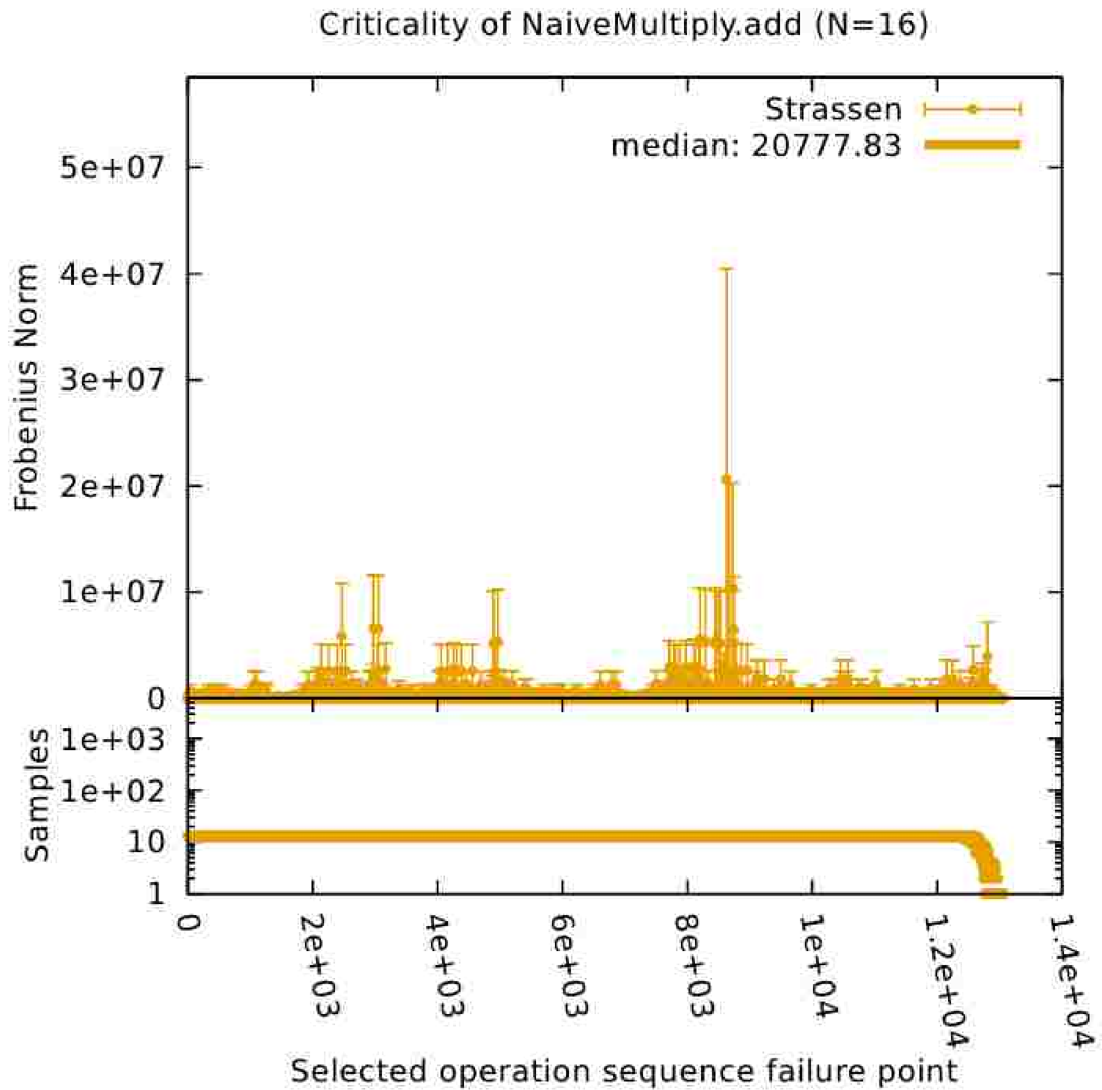


Figure 5.19: **Expanded Frobenius Norm Strassen's Criticality Assessment Results on NaiveMultiply.add (N=16).** See Figure 5.11 (page 79) for details.

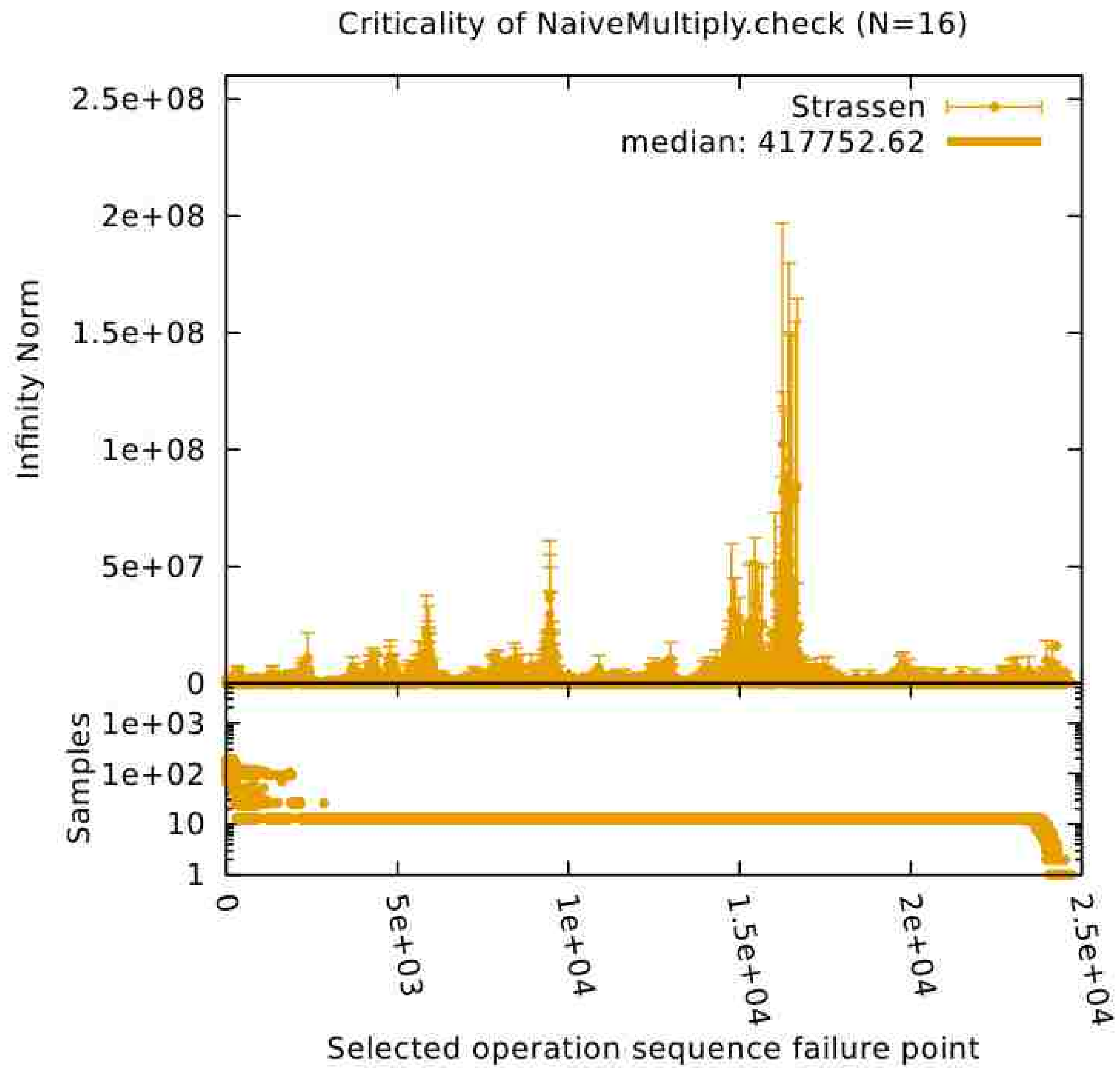


Figure 5.20: **Expanded Infinity Norm Strassen's Criticality Assessment Results on NaiveMultiply.check (N=16).** See Figure 5.11 (page 79) for details.

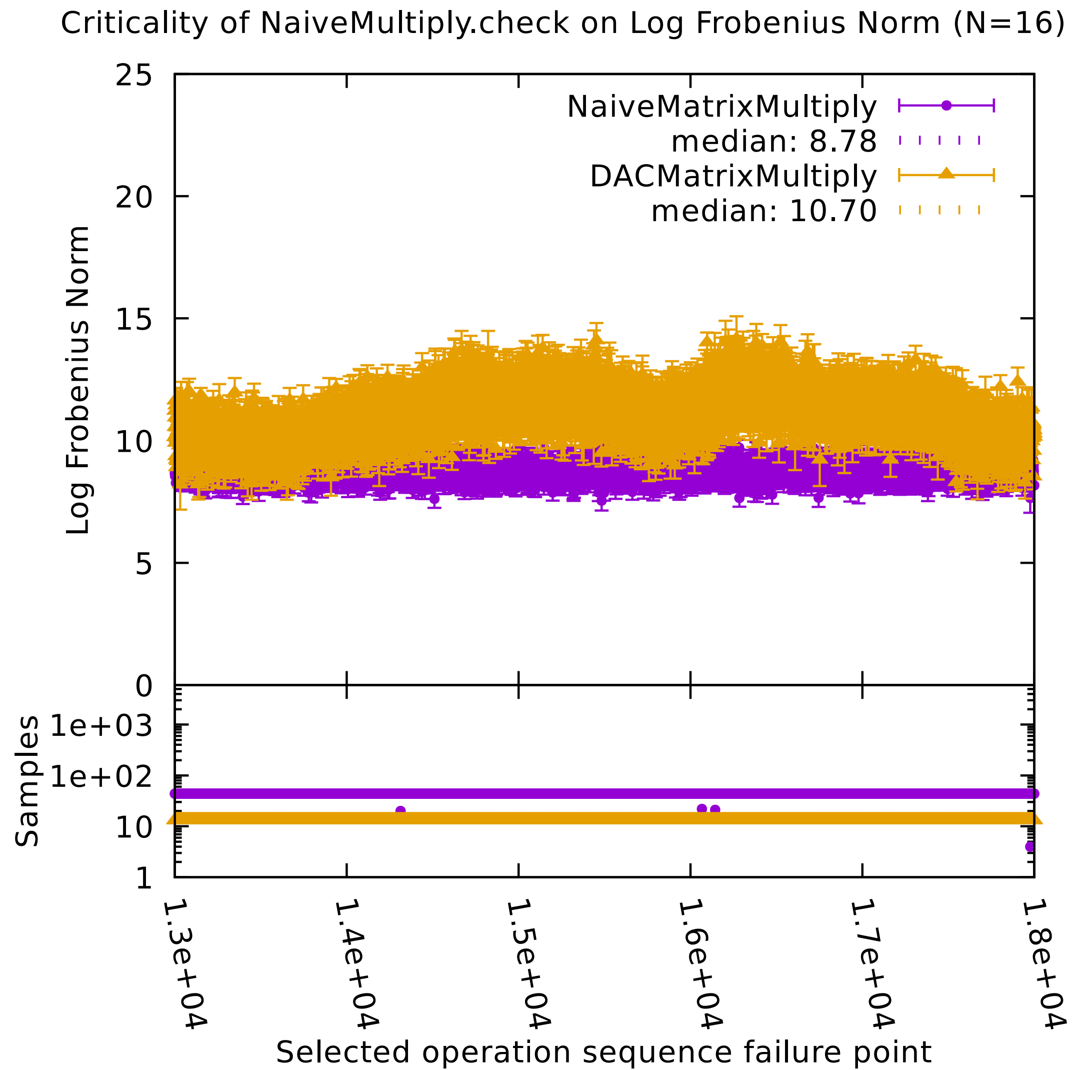


Figure 5.21: **Focused Log Frobenius Norm Criticality Results.** Log Frobenius norm structures continue to persist at larger input scales. See text for details. Reprinted from [1] with permission.

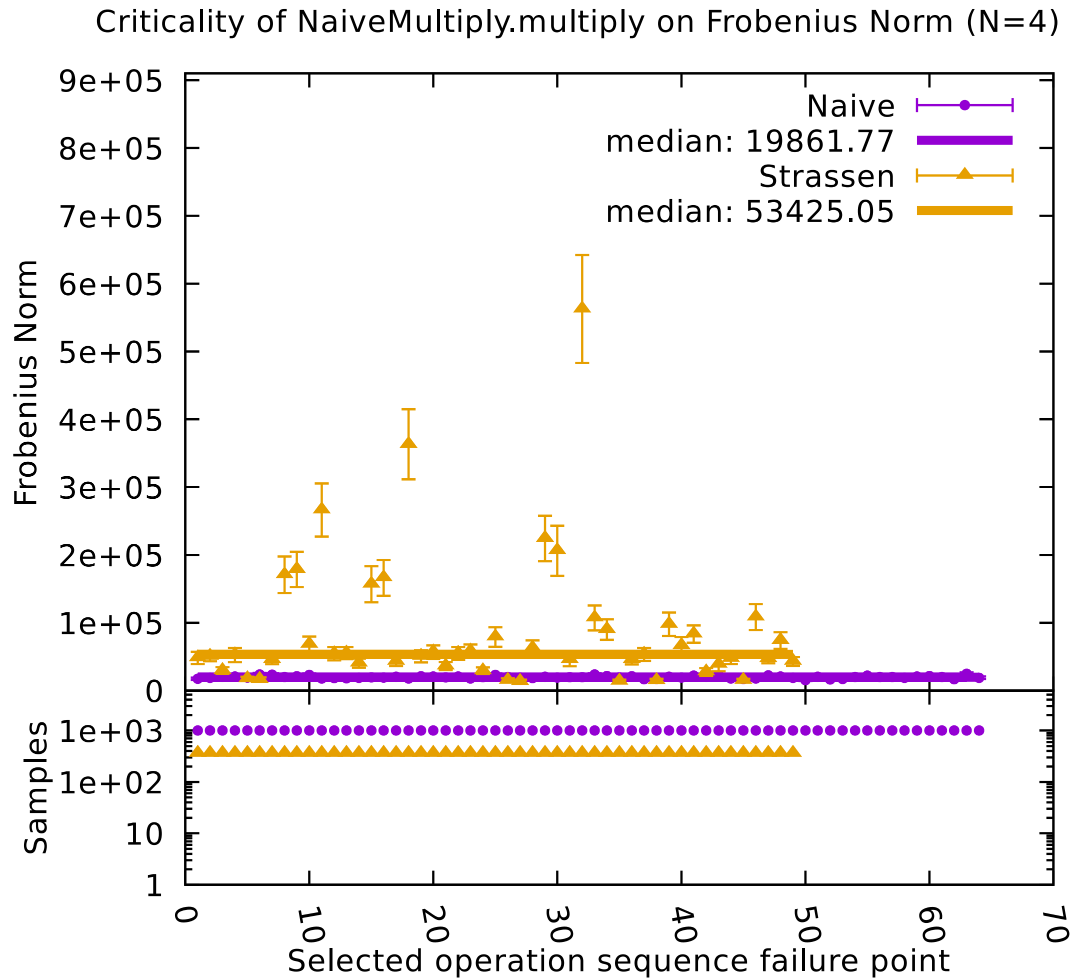


Figure 5.22: **Matrix Multiplication Proxy Method Criticality Assessment Results on Input Size N=4.** Criticality results for both naive and Strassen's algorithms on the scalar multiplication failure interface using *Frobenius norm* error measure at matrix input size N=4. See text for details. Reprinted from [1] with permission.

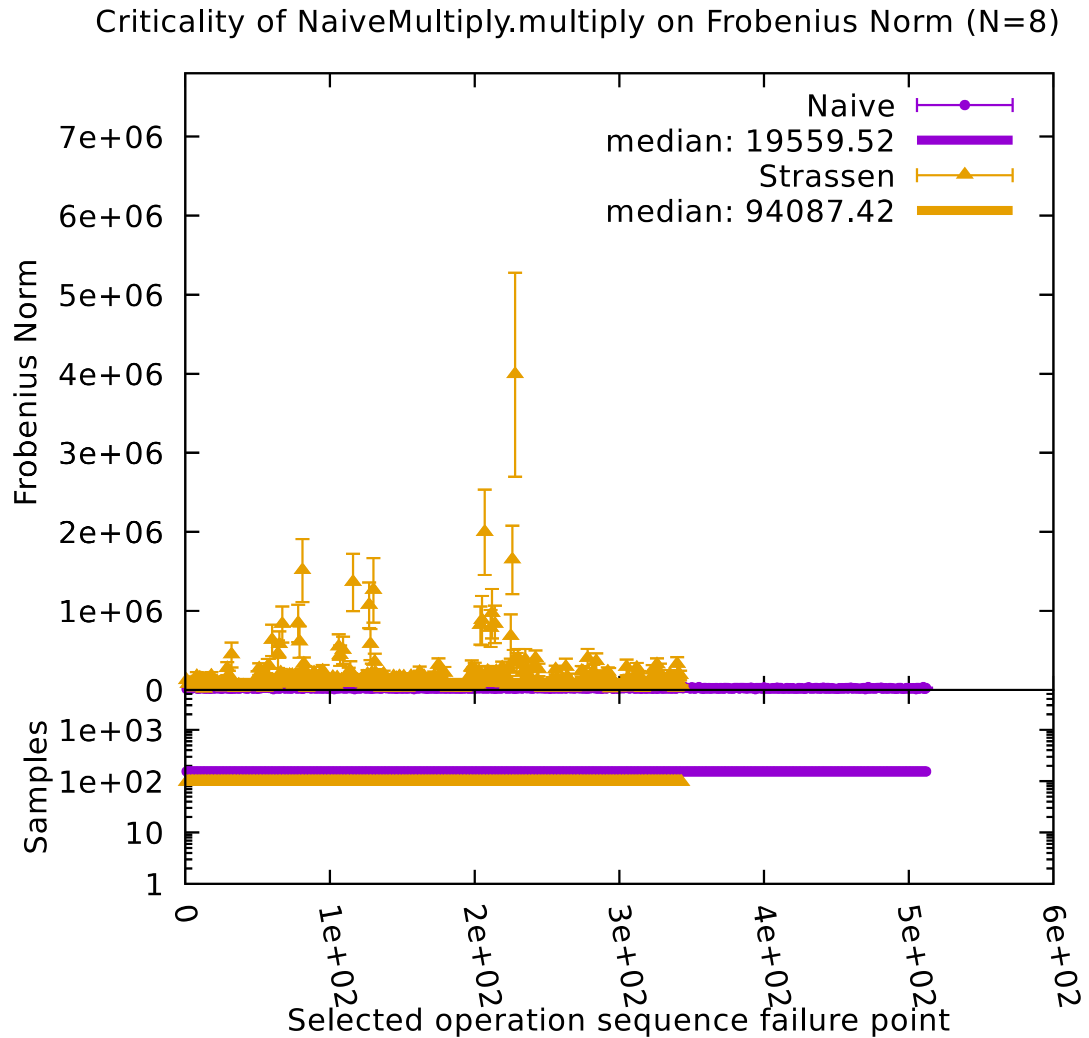


Figure 5.23: **Matrix Multiplication Proxy Method Criticality Assessment Results on Input Size N=8.** Criticality results for both naive and Strassen's algorithms on the `scalar multiplication` failure interface using *Frobenius norm* error measure at matrix input size N=8. See text for details. Reprinted from [1] with permission.

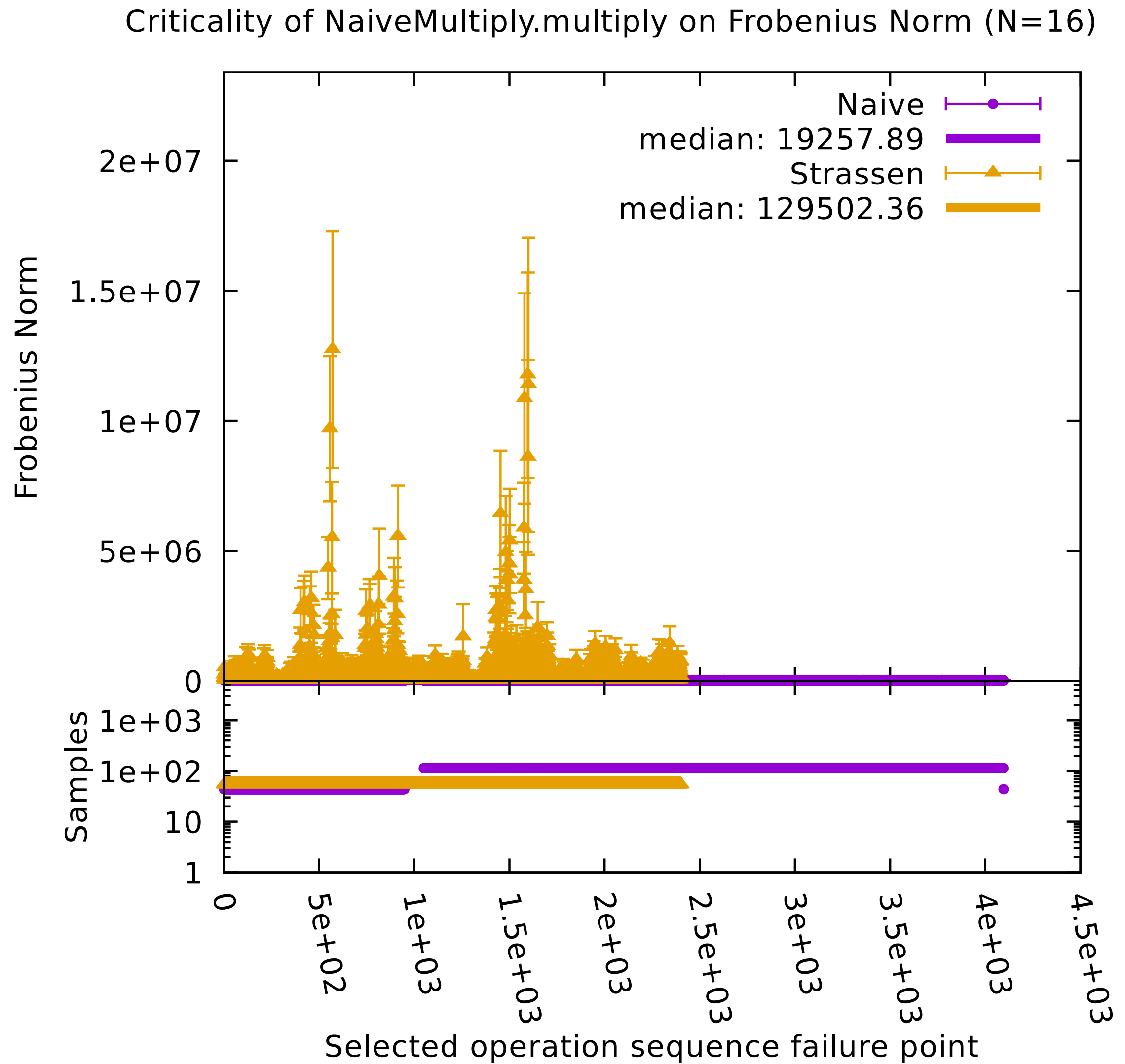


Figure 5.24: **Matrix Multiplication Proxy Method Criticality Assessment Results on Input Size N=16.** Criticality results for both naive and Strassen's algorithms on the `scalar multiplication` failure interface using *Frobenius norm* error measure at matrix input size N=16. See text for details. Reprinted from [1] with permission.

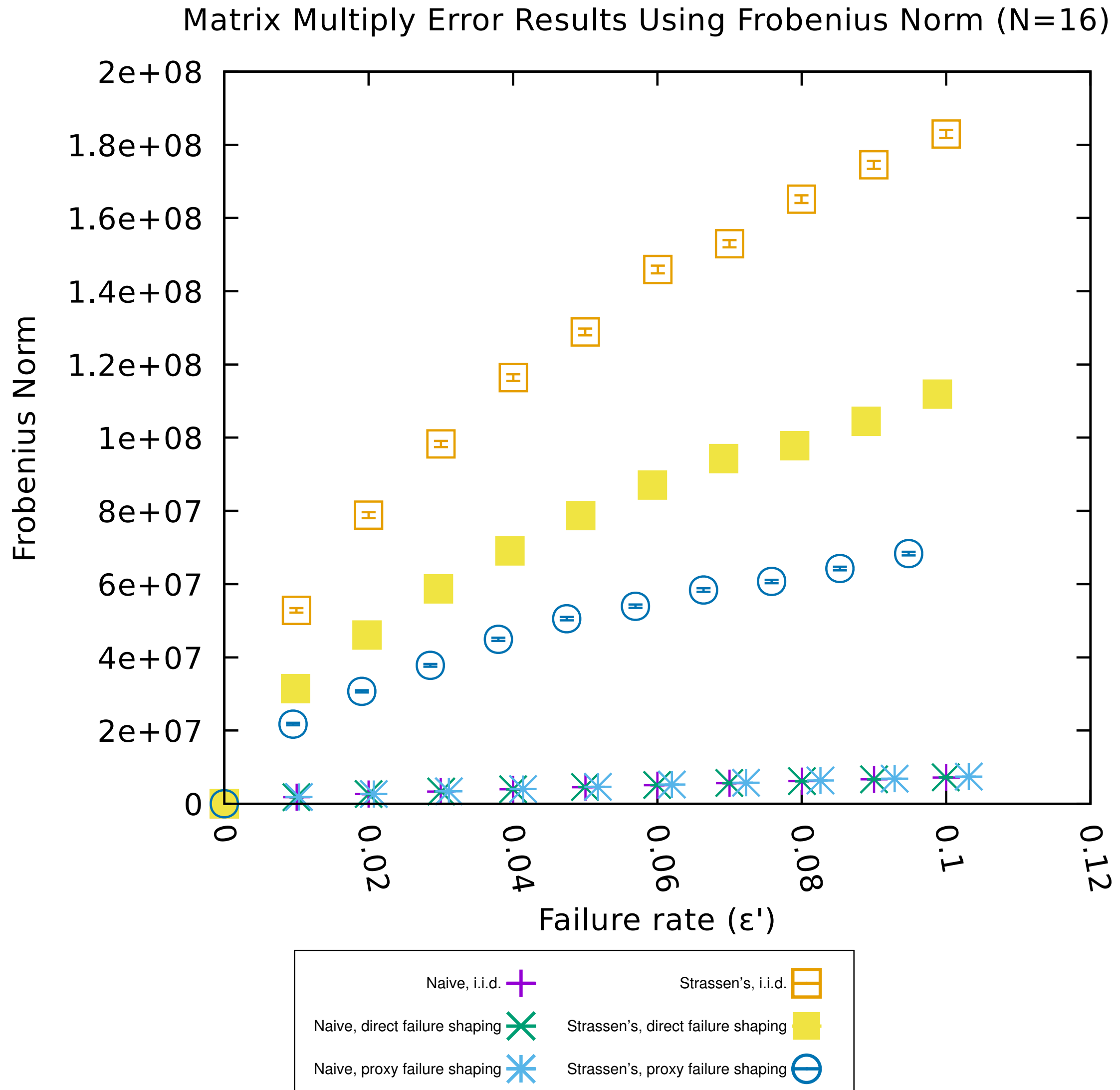


Figure 5.25: **Matrix Multiplication Proxy Failure Shaping Results on Frobenius Norm.** Average error rates for both the naive and Strassen's algorithms assessed using *Frobenius norm* error measure at input size 16. Assessments were performed on both algorithms using a baseline i.i.d. failure model, the direct failure shaping technique, and the proxy method failure shaping technique. We see error reductions between 38% and 63%. See text for details. Reprinted from [1] with permission.

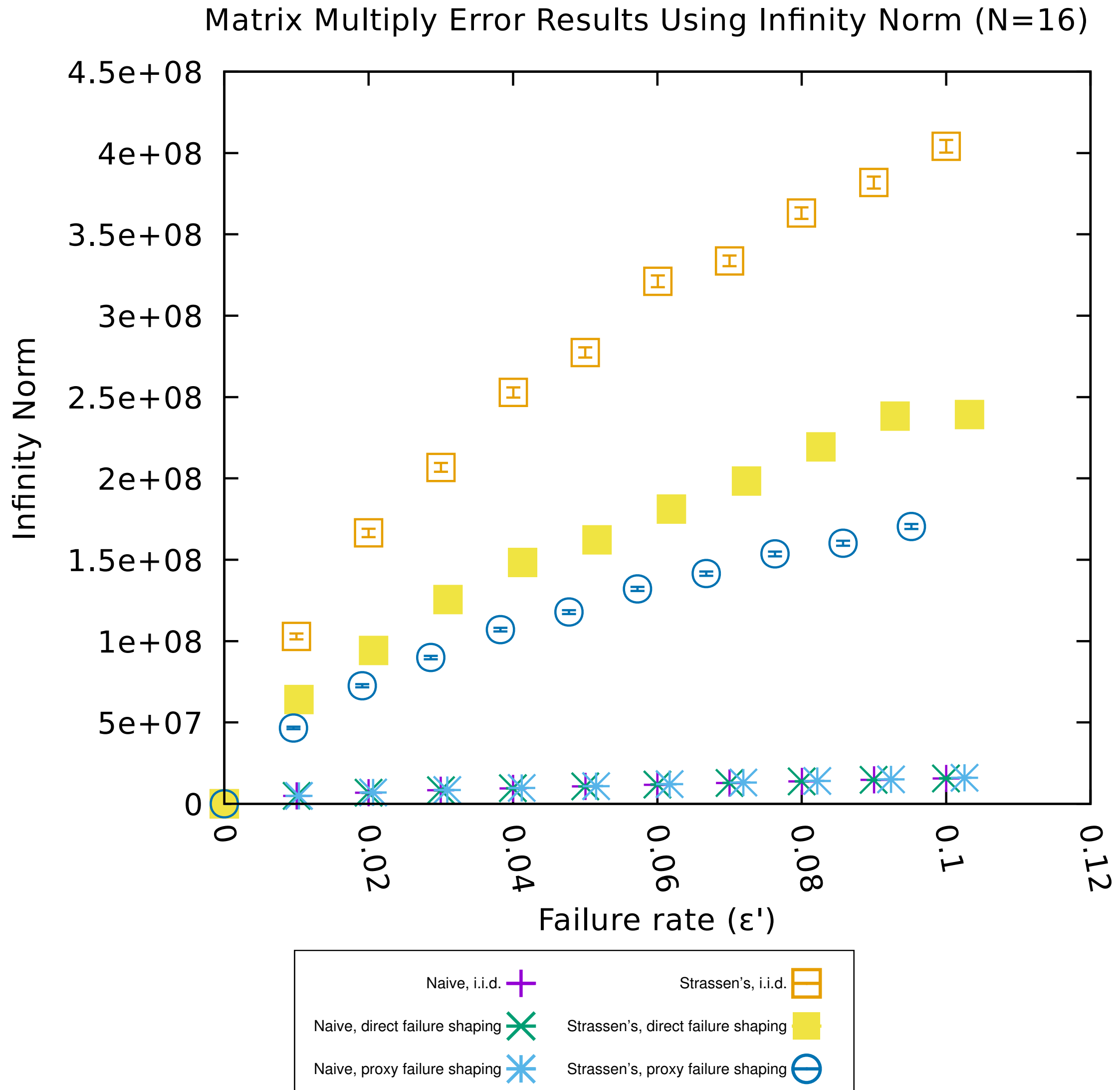


Figure 5.26: **Matrix Multiplication Proxy Failure Shaping Results on Infinity Norm.** Average error rates for both the naive and Strassen's algorithms assessed using *infinity norm* error measure at input size 16. Assessments were performed on both algorithms using a baseline i.i.d. failure model, the direct failure shaping technique, and the proxy method failure shaping technique. See text for details. Reprinted from [1] with permission.

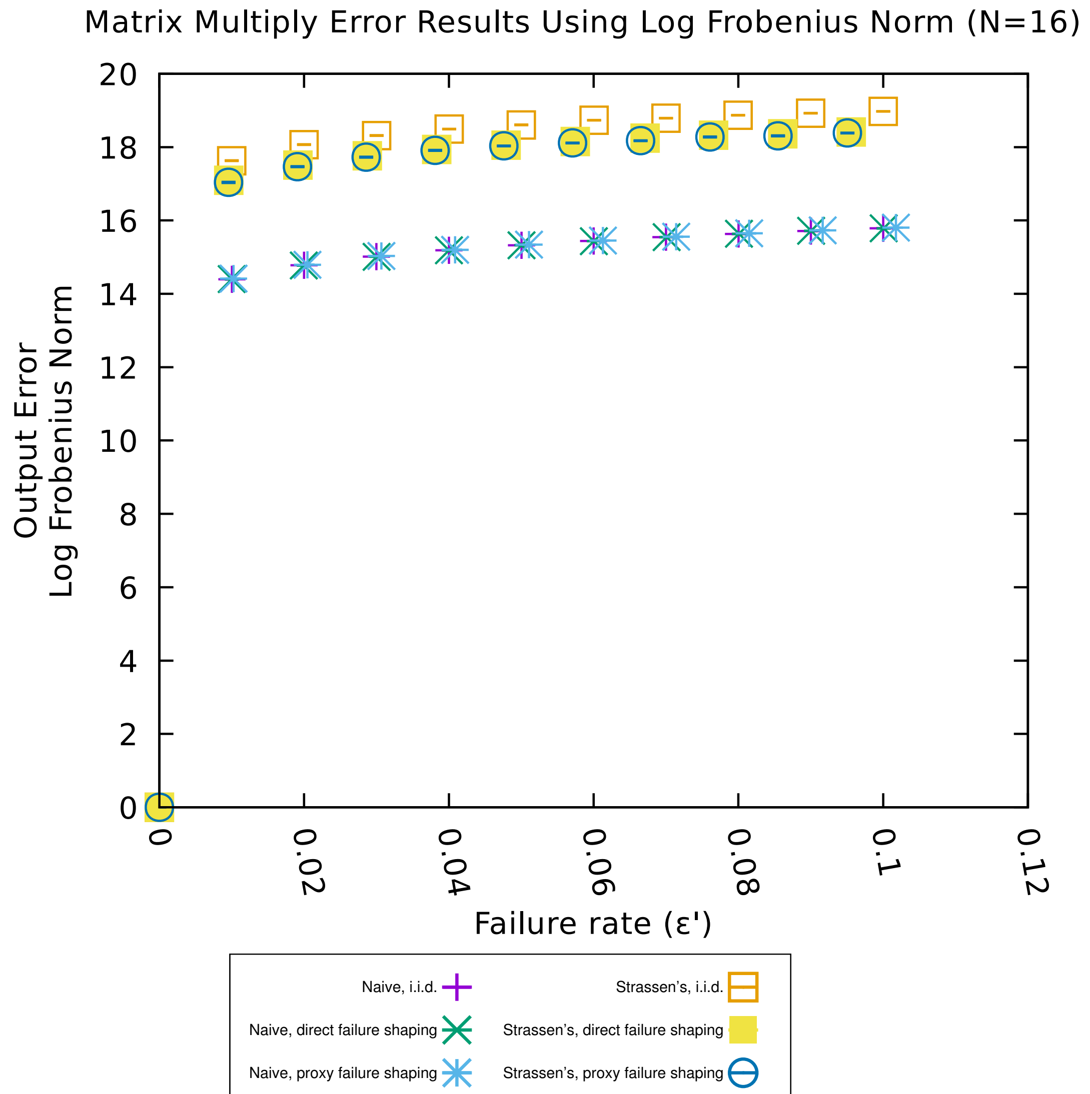


Figure 5.27: **Matrix Multiplication Proxy Failure Shaping Results on Log Frobenius Norm.** Average error rates for both the naive and Strassen's algorithms assessed using *frobenius norm* error measure at input size 16. Assessments were performed on both algorithms using a baseline i.i.d. failure model, the direct failure shaping technique, and the proxy method failure shaping technique. See text for details. Reprinted from [1] with permission.

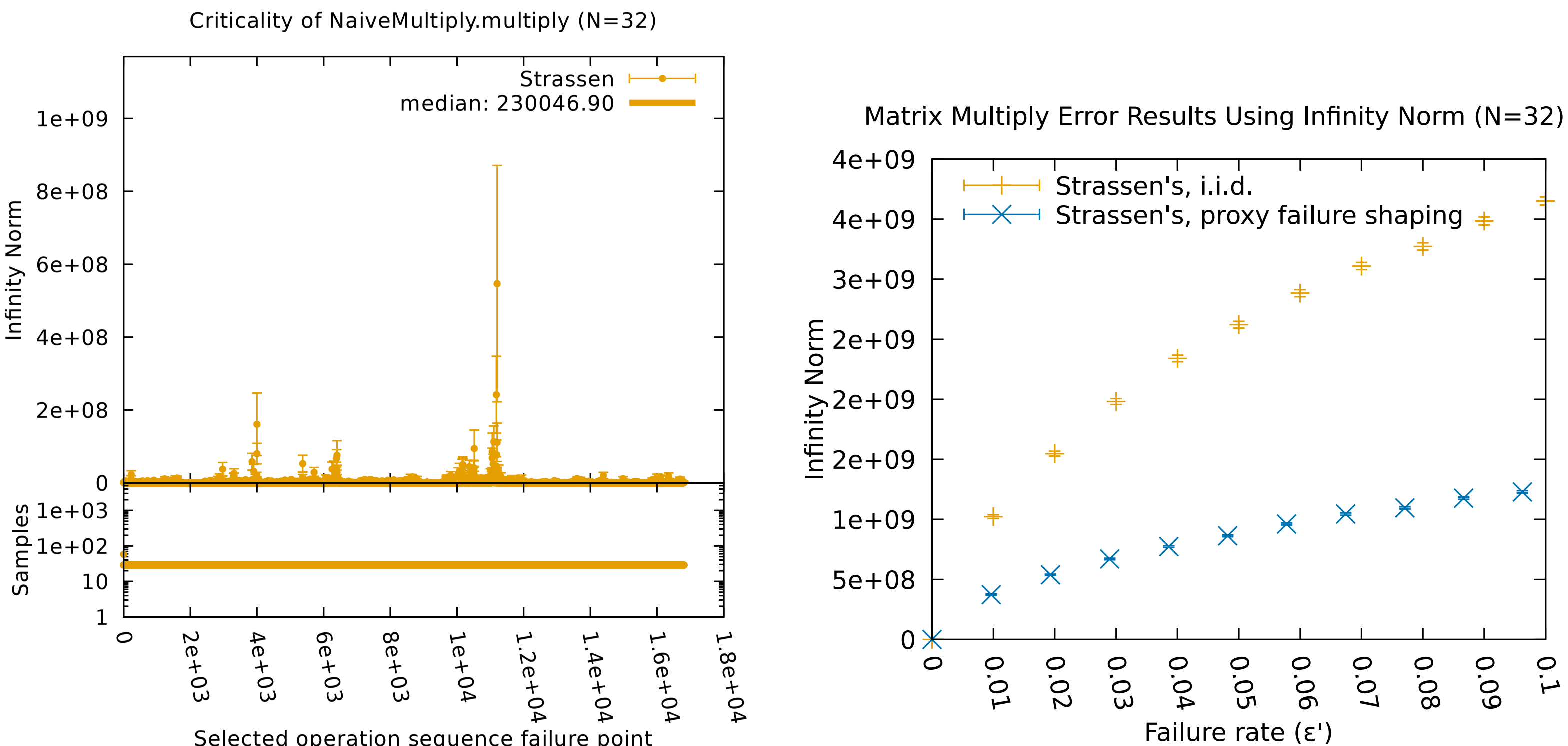


Figure 5.28: **Proxy Criticality Assessment and Failure Shaping Results on Input Size 32.** On the left I present the criticality assessment for proxy failure interface `scalar multiply` on the *infinity norm* error measure. It presents similar structures at input size 32 as at smaller input scales. On the right I present proxy failure shaping results for size 32 without direct failure shaping results. See text for details. Reprinted from [1] with permission.

Chapter 6

Scalable Robustness

6.1 Author Contribution Statement

I am the lead author of the paper underlying this chapter under the supervision of Dave Ackley (Associate Professor Emeritus, UNM Computer Science). I outlined, developed, and wrote the theoretical framework and proofs provided in this chapter. Dr. Ackley aided in the development of appropriate figures and tables for this chapter.

6.2 Publication Notes

Citation: T. B. Jones and D. H. Ackley, "Scalable robustness," *2016 46th Annual IEEE/ IFIP International Conference on Dependable Systems and Networks Workshop*, pp. 31-38, IEEE, 2016.

Editor: Lisa O'Conner

Received: March 28, 2016

Accepted: April 20, 2016

Published: June 28, 2016

Copyright: © 2016 IEEE

Formatting: Substantial parts of this chapter are derived directly from the original text, however some parts have been edited for clarity, correctness, or modified to fit with this dissertation. Figures and tables are mostly unchanged, though they have been modified to fit with this dissertation.

Data Availability Statement: Data is available at
`https://github.com/ThomasBJones2/Robustness_Dissertation_Data`.

Funding: No funding was provided for this work.

Competing Interests: The authors have declared that no competing interests exist.

Supporting Information: All pertinent information in from the original work is contained in this chapter.

6.3 Introduction

Using insights gained from Chapter 4, in this chapter I present a formal theory of *scalable robustness* oriented around the idea that scalably robust programs should have an output error that approaches zero as the internal i.i.d. failure rate of components running the program approaches zero and program input size approaches infinity.

6.4 Scalable Robustness Defined

We ask if an algorithm is scalably robust in a particular *context*, which is a combination of an *input generator*, a *failure model*, and an *error measure* as in Chapters 4 and 5. Scalable robustness is a property of a (program, context)[1] pair, which I call an *assessment*. In a scalably robust assessment the final output error goes to zero as the component failure rate vanishes, even as the input size grows without bound. As a result, in a scalably robust assessment we can bound output error rates *over all large scales* by bounding per component failure rates to a constant.

6.4.1 Average-Case Scalable Robustness

Though they fail to capture more complex failures such as device aging or attack, *i.i.d.* (independent, identically distributed) errors are a simple and physically plausible model of background noise and hence I treat them as the *average-case* failure model.

Definition 1. (A, err) *is* average-case scalably robust, *written* $\mathrm{ACSR}_{iid}(A, err)$, *if:*

$$\lim_{(N,\epsilon)\to(+\infty,0^+)} \left(\mathop{\mathbf{E}}_{i\in I_N} [err(A_\epsilon(i), A(i))] \right) = 0$$

where N *is an input size,* A *is an algorithm,* $\mathrm{err}(O_\epsilon, O_c)$ *is a context-dependent error measure defined over incorrect and correct algorithm outputs,* ϵ *is the i.i.d. failure rate of* A*'s fallible operations,* $\mathrm{A}_\epsilon(\mathrm{i})$ *is* A*'s output given input* i *and* ϵ *while* $A(i)$ *is* A *run without failures, and* $\mathrm{I_N}$ *is the set of all possible inputs of size* N*.*

We assume the context of A is chosen so that $\mathbf{E}_{i\in I_N}[err(A_\epsilon(i), A(i))] \in [0, 1]$ is uniquely defined at every (N, ϵ) in $N \geq 0$ and $0 \leq \epsilon \leq 1$.

[1] Since in this chapter I hold the input generator and fault model constant, I instead refer to each context only by the error measure it employs.

We can determine which assessments are $ACSR_{iid}$ by examining the behavior of the expectation from Definition 1 for all ϵ near 0. To do this we produce an upper bound on the maximum growth rate of output error, B_δ, for all $\epsilon \leq \delta$, or, similarly, a lower bound on the minimum growth rate B'_δ.

Notice that if the upper bound on an assessment's expectation for $\epsilon \leq C$ is:

$$B_C \in O(f(\epsilon)) \tag{6.1}$$

where C is a constant and $\lim_{\epsilon \to 0^+} f(\epsilon) = 0$, then the assessment is $ACSR_{iid}$.

Additionally, if

$$B'_\delta \in \Omega(f(\epsilon)g(N)) \tag{6.2}$$

where $\lim_{N\to\infty} g(N) = \infty$ and $\epsilon > 0 \to f(\epsilon) > 0$ then the assessment is not $ACSR_{iid}$ since a path of $f(\epsilon) = 1/g(N)$ violates Definition 1.

An immediate consequence of this idea is the following theorem concerning strict correctness error (SCE):

Theorem 1. *(A, SCE) is $ACSR_{i.i.d.}$ for all programs A if the run time of the program is $O(f(N)) \in \omega(c)$ and each operation is armored with $\Omega(lg(f(N)))$-modular redundancy.*

Proof. We start by bounding the probability of failure on a single redundant operation. We do this by noticing that each operation will perform correctly if more than half of the redundant components that calculate the operation also perform correctly. For the number of redundant operations we choose $g(n) = 24lg(f(N))$

Using a multiplicative Chernoff bound, we know that if X is a Bernoulli variable with mean μ and $\delta \geq 1$:

$$P[X \geq (1+\delta)\mu] \leq e^{-\delta\mu/3}$$

Next we let $\delta = 1/(2\epsilon) - 1$ and we remember that $\mu = \epsilon g(N)$. Using these we obtain:

$$P[X \geq 1/(2\epsilon)\epsilon g(N)] \leq e^{-(1/(2\epsilon)-1)\epsilon g(N)/3}$$
$$P[X \geq (1/2)g(N)] \leq e^{-(1/2-\epsilon)g(N)/3}$$

However if we hold $\epsilon \leq 1/4$ (and notice that ϵ is positive in the exponent of e) we then obtain that:

$$P[X \geq (1/2)g(N)] \leq e^{-(1/2-1/4)g(N)/3} = e^{-2(g(N)/24)} = e^{-2lg(f(n))} = 1/(f(N)^2)$$

However, there are $f(N)$ operations in A and we limited the probability of the redundant operations failing to $1/(f(N)^2)$. Therefore, the expected number of failures is $f(N)/(f(N)^2)$ or $1/f(N)$ which goes to zero as N goes to infinity. Since the maximum SCE is bound to be smaller than the expected number of failures when $1/f(N)$ is less than 1 this proves Theorem 1. □

This theorem complements other results that show that $O(lgf(N))$ redundancy is a *lower bound* on the number of redundant operations needed to produce deterministically secure execution for some functions [99, 100].

Theorem 1 relies on the idea that failures will occur under an i.i.d. model, but that the per-operation failure rate will decrease with the increasing size of the computation by increasing the number of components performing each operation. However, to make the redundant computation viable, it is likely that redundant operations be closely *collocated*, leading to a higher probability of coordinated failures. Such coordinated failures, which would impact the final outcome of the program, can be viewed as producing an i.i.d. failure behavior 'above' the modularly redundant and thus 'deterministically secure' $\Omega(lg(n))$-modularly redundant components.

It is this tendency for coordinated failures to occur when operations are collocated in space that motivates the assumption that some component of programmatic error rates remain at a very small constant above zero as programs approach large scales. In the worst case, attackers may use coordinated failures to drive up program error rates.

6.4.2 Worst-Case Scalable Robustness

I.i.d. faults are a natural simplifying assumption, but we expect higher-order correlated failures to occur as well. In the most general case, no defense is possible against an unlimited supply of adversarially-chosen faults; here we define *worst-case scalable robustness* in terms of adversarial faults restricted only so that their total number is some given proportion of the program size $f(N)$:

Definition 2. *(A, err) is* worst-case scalably robust*, written* WCSR(A, err)*, if:*

$$\lim_{(N,\epsilon)\to(+\infty,0^+)} ArgMax_{i\in I_N, f\in F_{f(N),\epsilon}}\ err(A_\epsilon(i), A(i)) = 0$$

with N, A*, and* err(O_ϵ,O_c) *as in Definition 1.* f*, called a* fault pattern [2]*, is a Boolean vector indicating which of the fallible computational steps do fail in any given case,*

and $F_{f(N),\epsilon}$ is the set of all fault patterns that each contain no more that $\epsilon f(N)$ true values. Finally, $A_f(i)$ is the algorithm A run with operations faulting as called for in fault pattern f on input i while $A(i)$ is the same program run without failure on input i.

We use ArgMax in Definition 2 to represent an omniscient attacker causing maximum damage under the given $f(N)$ and ϵ.

As implied by their names, we can show that any algorithm-error measure pair that is *WCSR* is also $ACSR_{iid}$:

Theorem 2. *If* $\mathrm{WCSR}(A, err)$ *then* $\mathrm{ACSR}_{iid}(A, err)$ *for all algorithms with growth rate* $g(n) \in \omega(c)$. [2]

Proof. Our error measures send outputs to a numeric values in $[0, 1]$. Therefore, if (A, err) are *not* $ACSR_{iid}$ there is some path p of ϵ and N going to 0^+ and $+\infty$ such that the average error from Definition 1 is at or above some constant $C_1 > 0$ past infinitely many points along $p : \epsilon = g(f(N))$ where $f(n) \in \omega(c)$ is the number of fallible operations in the program and $g(f(n)) \to 0$ as $N \to +\infty$.

However, notice that according to Chebyshev's inequality [101] if X is a random variable with expected value μ, variance σ, and $k > 0$ then:

$$
\begin{aligned}
P(|X - \mu| \geq k) &\leq \sigma^2/k^2 \\
P(X \geq \mu + k) &\leq \sigma^2/k^2
\end{aligned}
$$

However, for the sum of $f(N)$ i.i.d. failure variables with failure rate ϵ, we know that $\mu = \epsilon f(N)$ and that $\sigma^2 = \epsilon(1 - \epsilon)f(n)$, Thus:

[2] [3] contained a flawed version of this proof; it is corrected here.

$$P(X \geq \epsilon f(n) + k) \leq \epsilon(1-\epsilon)f(n)/k^2 \leq \epsilon f(n)/k^2$$

Now, suppose that I let $k = f(n)^{1/2}$ which is greater than 0 since $f(n)$ is the number of operations in A. If we do this we obtain:

$$P(X \geq \epsilon f(n) + \sqrt{f(n)}) \leq \epsilon$$
$$P(X \geq f(n)(\epsilon + 1/\sqrt{f(n)})) \leq \epsilon$$

However, ϵ is going to zero. This means that the probability that any item in A will be drawn from above the path $p : \epsilon + 1/\sqrt{f(n)} = g(f(n)) + 1/\sqrt{f(n)}$ is a value that approaches zero as $(N, \epsilon) \to (+\infty, 0)$. Thus, because an average cannot lie above all values in its data set, there must be some specific combination of failures and input with less than $g(f(n)) + 1/\sqrt{f(n)}$ total failures along infinitely many points on p where $err(A_f(i)) > C_2 > C_1 - \epsilon > 0$ when ϵ drops permanently below C_1.

Thus, if $\neg\text{ACSR}_{iid}(A, err)$ then $\neg\text{WCSR}(A, err)$, proving Theorem 2. □

6.5 Scalable Robustness on Pairwise-Comparison Sorting

In applying the theory of scalable robustness to sorting I adopt many of the conventions of Chapter 4. As in that chapter, I presume an input generator that produces permutations of N distinct data values, and a fault model that allows only data item comparisons to fail. Though extending the input generator is likely unproblematic—to add duplicate values, for example—generalizing the fault model significantly would

require a more detailed machine model in which to express the algorithm implementations, a step I leave for future research. I employ four sorting algorithms—bubble sort, quicksort, merge sort, and 'round robin sort'—in this case study of $ACSR_{iid}$ and *WCSR*.

6.5.1 Algorithms Old and New

As in Chapter 4 the bubble sort, quicksort, and merge sort algorithms we explore are largely standard. Although bubble sort implementations (e.g., [94]) commonly perform $N - 1 - P$ comparisons on pass P through the list, our implementation—which we call *full bubble sort* (FBS)—makes all $N-1$ neighbor-neighbor comparisons on every pass. My merge sort implementation (MS) is traditional (see, e.g., [34]), and my quicksort (QS) [93] uses the first item in the list as a pivot—a poor choice in general but harmless given my adopted input generator.

I call my final algorithm *round robin sort* (*RRS*). It is inspired by the round robin tournament [102], and, while the idea of using round robin comparisons in sorting is fairly common [103], I haven't found this exact algorithm in the literature. RRS steps through each list item comparing that item to all other items. It counts the number of times that item is greater than other items, and places it in a bin based on that count. The items are then returned in the order of the bins. Any bin that has more than one item returns items to the output list in the same order they appear in the input list.

$ACSR_{iid}$ Error Growth Rates

	Merge	**Quick**	**Bubble**	**Round Robin**
Strict Correctness	$\Omega(\epsilon N lg N)$	$\Omega(\epsilon N lg N)$	$\Omega(\epsilon N)$	$\Omega(\epsilon N)$
Max Displacement	$\Omega(\epsilon N)$	$\Omega(\epsilon N)$	**?**	**?**
Spearman's Footrule	**?**	$O(\epsilon)$	**?**	$O(\epsilon)$

Table 6.1: **Summary of $\boldsymbol{ACSR_{iid}}$ Results.** Where known, each (sorting algorithm, error measure) assessment is marked with a bound on its expected error measure growth rate—$O()$ upper bounds for assessments proven $ACSR_{iid}$, and $\Omega()$ lower bounds for those proven *not* $ACSR_{iid}$. Those marked **?** are currently unproven, though empirical data (e.g. Fig. 6.1) suggests they are all $ACSR_{iid}$. Reprinted from [3] with permission.

6.6 Average-Case Scalable Robustness on Sorting Algorithms

$ACSR_{iid}$ illuminates the tension between algorithmic efficiency and robustness. Table 6.1 provides lower bounds on the average error growth rate near 0 (marked with Ω) for each assessment that is provably not $ACSR_{iid}$ and, similarly, upper bounds (marked with O) for each assessment that is provably $ACSR_{iid}$. Assessments that are currently unevaluated are marked with a '**?**', though I believe them to be $ACSR_{iid}$ due to my empirical results. I present proofs for Table 6.1 below.

Theorem 3. (MS, SCE) *is not* ACSR$_{iid}$.

Proof. Notice that if any comparison in the merge sort algorithm fails then some smaller item will be trapped above a larger item leading to an incorrect sort. Further merge sort executes $\Theta(N lg N)$ comparisons. Therefore $B'_{1/N lg N} \in \Omega(\epsilon N lg N)$ since $\epsilon = 0$ has no error while there be a failure 1/2 time if $\epsilon \in \Omega(1/N lg N)$.

□

Similar proofs can be constructed for quicksort and bubble sort. In the Bubble Sort proof only the final pass is considered since any incorrect comparison there will produce an incorrectly sorted output list.

Theorem 4. *(RRS, SCE) is not* ACSR_{iid}.

Proof. Sometimes, RRS can produce correct output even if one or more items end up in the wrong bin(s), but the chance of an incorrectly sorted output is at least 1/2 in all such cases. Additionally, items in the top quartile of the list should be declared 'greater than' other items at least three times as often as they are declared 'less than' another item. Therefore, if $\epsilon \geq 4/N$ each item in the top quartile will experience at least 4 failed comparisons 1/2 the time. Each of these items with 4 failed comparisons will be in the wrong bin with probability at least 1/4 due to the asymmetry in the number of 'less than' and 'greater than' comparisons they experience. This means that $\epsilon \geq 4/N$ leads to an expected output error of at least 1/16 or that $B'_{4/N} \in \Omega(\epsilon N)$.

□

Next I show that both linearithmic algorithms are not $ACSR_{iid}$ when assessed by max displacement error.

Theorem 5. *(MS, MDE) is not* ACSR_{iid}.

Proof. Let the larger item in a failed comparison of merge sort be ℓ and the smaller item be s. When a comparison fails, s will always be trapped above ℓ in any future lists, including the final list. Further, notice that the $disp(s) \geq dist(s, \ell) - failed(\ell)$, where $disp(s)$ is the displacement of s, $dist(s, \ell)$ is the distance between s and ℓ in a correctly sorted list, and $failed(\ell)$ is the number of times ℓ experiences a failed comparison with an item that is smaller than ℓ but not s.

In the lowest-depth merges of merge sort, ℓ and s are random list items. This means that the expected distance between them is $N/2$. Additionally, $failed(\ell) \leq \epsilon N$ in expectation because each item is compared with at most N other items. Therefore, $disp(s) \geq N/2 - \epsilon N$ in expectation if ℓ and s are in the first, lowest, merge. Thus, the expected error when one of the lowest-depth merges has a failed comparison is at least $(1/2 - \epsilon)N$. Lastly, the probability that one of the lowest merge comparisons will fail is at least 1/2 when $\epsilon = 1/N$.

By holding $1/N \leq \epsilon \leq 1/4$ (which is possible for all $N > 4$) an output error of approximately 1/8 can always be obtained in expectation implying that $B'_{1/N} \in \Omega(\epsilon N)$. □

Theorem 6. *(QS, MDE) is not* ACSR_{iid}.

Proof. Notice that if any comparison in the quicksort algorithm fails on an item x and pivot p then $disp(x) \geq dist(x, p) + 1 - disp(p)$ since x will always be placed above p in the output list. If the failed comparison occurs during the first pass of the algorithm then

$$E[disp(x)] \geq E[dist(x, p) - failed(p)]$$

since x and p are chosen by random processes $E[dist(x, p)] = N/2$ while $E[failed(p)] \leq \epsilon N$ since p is compared to at most N items. From here we can see that $B'_{1/N} \in \Omega(\epsilon N)$ using arguments similar to those in Theorem 5.

□

Next I prove that quicksort is $ACSR_{iid}$.

Theorem 7. *(QS, SFE) is* ACSR_{iid}.

Proof. To prove that Definition 1 holds for (QS,SFE) we notice that it is sufficient to show that:

$$E[SFE(QS_\epsilon(I_N), I_N)] \leq \epsilon \frac{O(N^2)}{\Theta(N^2)} \in O(\epsilon) \tag{6.3}$$

We will show this by bounding the *marginal expected error*—termed $M(x)$—for each list item as it flows through the quicksort algorithm. Throughout this proof will call the pivot at depth d, p_d, and the sublist sorted on that pivot S_{p_d}. Now, we will bound the expected marginal error introduced by failed comparisons on list item x when x is a *non-pivot.* If we call the final depth of x before it becomes a pivot f then as x flows through the algorithm, it will be a member of a number of sublists—S_{p_0}, S_{p_1},...,S_{p_f}, S_x. In each of these sublists, the *marginal unnormalized SFE* introduced by a failed comparison is bounded by $2|S_p|$—$|S_p|$ from x being misplaced above or below p and $|S_p|$ for each other item in the sublist being shifted up or down one space due to x's misplacement.

Since each comparison fails with a probability of ϵ and each list item is compared with each pivot exactly once before it becomes a pivot we can treat the failure of each of the comparisons between x and its pivot p as a Bernoulli variable with probability ϵ. Using $2|S_p|$ as a bound on the marginal error on each comparison with p we see that

$$M(x) \leq 2\epsilon \sum_{d=0}^{f} E[|S_{p_d}|] \tag{6.4}$$

We say that x participates in a *successful split* if x is sent to a sublist of length no greater than 1/2 of the previous list. Since the pivot and x are both random items in the sublist, the probability that x participates in a successful split at each level of the quicksort recursion is at least 1/2.

At a recursive depth of d, this means that we can bound the *expected length* of the sublist in which x appears. We know that the probability that x participates in q successful pivots at recursive depth d is at least $\binom{d}{q}(1/2)^d$. We also know that if x participated in q successful pivots then $|S_{p_d}|$ is at most length $N(1/2)^q$. Therefore, the length of sublist at recursive depth d is at most $\sum_{i=0}^{d}\binom{d}{i}(1/2)^d N(1/2)^i$. Using the binomial theorem this shows that $E[|S_{p_d}|] \leq (3/4)^d$.

Using Equation 6.4 and summing over infinitely many possible depths we can show that $M(X) \leq \epsilon 8N$. Therefore, the total expected *unnormalized* Spearman's footrule error is bound by $\epsilon O(N^2)$ since there are N items in the list. Finally, the normalization factor for Spearman's footrule error is $\Theta(N^2)$ and as such we obtain Inequality 6.3 and show $B_C \in O(\epsilon)$.

□

I also present Theorem 8 which relates $ACSR_{iid}$ under SFE to $ACSR_{iid}$ under MDE.

Theorem 8. $\text{ACSR}_{iid}(A, MDE) \rightarrow \text{ACSR}_{iid}(A, SFE)$.

Proof. Consider that

$$
\begin{aligned}
E[SFE(A_\epsilon(I_N), I_N)]O(N^2) \leq \\
O(N)E[MDE(A_\epsilon(I_N), I_N)]
\end{aligned}
\tag{6.5}
$$

Since unnormalized SFE is simply the sum of the displacement of every item in the list and the normalization factor of MDE is O(N). However, we know that Inequality 6.5 implies that $\text{ACSR}_{iid}(A, MDE) \rightarrow \text{ACSR}_{iid}(A, SFE)$.

□

This theorem was paired with the purported, but mistaken, theorem that (RRS, MDE) is $ACSR_{iid}$ in [3] to obtain that (RRS, SFE) is also $ACSR_{iid}$. We do know that $ACSR_{iid}$(RRS, SFE) is correct anyway, however, because of Theorem 2 combined with Theorem 14 from the next chapter, concerning (RRS, SFE) behavior in the worst case.

Figure 6.1 presents data suggesting that the scalably robust assessments' average-case output error grows at a consistent rate, even as input sizes increase. Alternatively, assessments that are *not* scalably robust present ever-sharper 'knees' as input size grows. Each panel of Figure 6.1 shows the i.i.d. average-case error behavior of the sorting algorithms under SCE (left), MDE (middle), and SFE (right) for lists of size 100, 1000, 10000, and 100000. Each data point in the panels represents the error measure at a given list size and ϵ, averaged over the number of runs shown. Notice how all four algorithms experience a knee as ϵ goes from 0% to 0.1% under SCE, while only quicksort and merge sort experience such a knee under MDE. Finally, no algorithm experiences a knee under SFE. Assessments that are not scalably robust jump quickly from mostly correct to mostly incorrect when input sizes are large enough.

6.7 Worst-Case Scalable Robustness on the Sorting Algorithms

A synopsis of my theoretical exploration of *WCSR* on the sorting algorithms can be found in Table 6.2 (page 119).

Theorem 2 shows that none of the four algorithms is *WCSR* when assessed by SCE because none are $ACSR_{iid}$ under SCE. The same method proves that merge sort and quicksort are not *WCSR* under MDE. In Theorems 9 and 10 we show that neither of the quadratic algorithms is *WCSR* in an assessment with MDE.

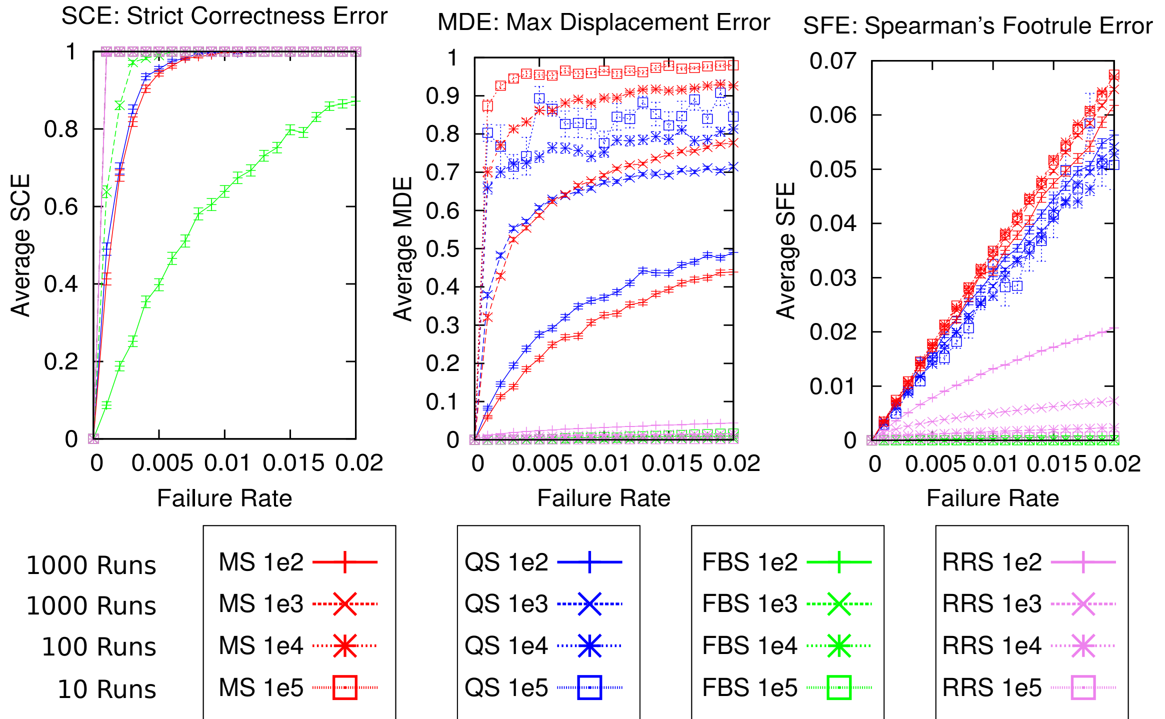


Figure 6.1: **Summary of Empirical Results.** Measured average error rates for merge sort (**MS**), quicksort (**QS**), full bubble sort (**FBS**), and round robin sort (**RRS**), assessed according to strict correctness error (**SCE**, left), max displacement error (**MDE**, middle), and Spearman's footrule error (**SFE**, right, note changed y scale), plotted vs the failure rate and at input sizes from 100 to 100,000 (**1e2..1e5**). See text for details. Reprinted from [3] with permission.

Theorem 9. *(FBS, MDE) is not* WCSR.

Proof. Consider the case where the input list is reverse sorted. Suppose an attacker flips the $(N/2)^{th}$ comparison on each of the N passes. Then, no item that starts in the bottom half of the list will ever move to the top half including the largest item which will have a displacement of at least $1/2$. Since we only fault N of the N^2 comparisons carried out by bubble sort, ϵ approaches 0 as N approaches ∞.

□

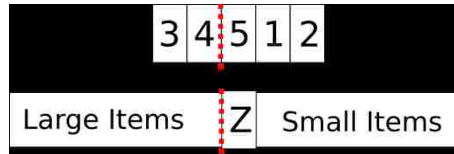


Figure 6.2: **Merge Sort WCSR Strategy:** The attacker faults comparisons with the largest item until the list is shaped with all the largest items excluding the largest item (Z) in the first half, and all the smallest items preceded by Z in the second half. The final merge compares items across the red-dashed line. Reprinted from [3] with permission.

Theorem 10. *(RRS, MDE) is not* WCSR*.*

Proof. Notice that if an attacker faults every comparison of the largest item in the list and no others then the max displacement error will be at least $1 - 1/N$ while epsilon follows the path of $\epsilon = O(1/N)$. □

In Theorems 11 through 13 we show that merge sort, quicksort and full bubble sort are not *WCSR* when paired with Spearman's footrule error.

Theorem 11. *(MS, SFE) is not* WCSR*.*

Proof. We start with list that is reverse sorted except that the largest item is in the second half of the list. The attacker then faults every comparison that the largest item participates in. At the final merge the list is split into two groups: the first group having the largest $N/2$ directly beneath the largest item sorted in order, and the second group having the smallest $N/2 - 1$ items in the list, preceded by the largest item which is either the first item in the second half of the list, or directly in the middle of the list. See Figure 6.2.

In the final merge, however, no item will switch places if no comparisons are faulted.

Thus the total Spearman's footrule error on the output list is at least $\frac{N(N-2)/2}{\Theta(N^2)}$ because each item is a distance of at least $(N-2)/2$ away from its correct position. As N approaches infinity this value is strictly greater than 0. This occurs with only $\Theta(lg(N))$ faulted comparisons (the failed comparisons of the largest item on each of $O(lgN)$ merges before the final merge). Thus a path of $\epsilon \in \Theta(1/lgN)$ exists such that Definition 2 is violated.

□

6.7.1 Quicksort—Spearman's Footrule Error

Theorem 12. *(QS, SFE) is not* WCSR*.*

Proof. Consider that if the middle item of the sorted list in quicksort is the first pivot and every one of the N comparisons that pivot participates in are faulted then all of the largest items will be in the first half of the output list and all the smallest items will be in the second half of the output list. As result the Spearman's footrule error will be above roughly 1/2 while epsilon follows a path of $\epsilon \in \Theta(1/lgN)$.

□

6.7.2 Bubble Sort—Spearman's Footrule Error

Theorem 13. *(FBS, SFE) is not* WCSR*.*

Proof. Using the same attacks as in Theorem 9 shows that (FBS,SFE) is not *WCSR*.

□

Finally, we show that round robin sort is *WCSR* when assessed in a context with Spearman's Footrule Error.

6.7.3 Round Robin Sort—Spearman's Footrule Error

Theorem 14. *(RRS, SFE) is* WCSR*.*

Proof. To prove that RRS is $WCSR$, we will use an inductive amortization analysis [104] to show that the amount of *unnormalized* Spearman's footrule error, referred to as $USFE$ throughout this proof, is bounded by a function that is proportional to the number of failed comparisons. Since the normalization factor of $USFE$ is $\Theta(N^2)$ and the number of comparisons in RRS is also $\Theta(N^2)$, this will suffice to show that (RRS, SFE) is worst-case scalably robust. Note that this proof relies on the fact that all items in the input list are distinct.

We begin with some definitions necessary for our proof:

Definition 3. *$Bin(k) = k$ is the bin of the kth list item when the list is correctly sorted and $Bin_{i,f}(k)$ is the bin of the kth list item when input i is sorted under some fault pattern f.*

Next we define the *drift* of items during a faulty run of the RRS algorithm in Definition 4.

Definition 4. *$D_{i,f}(X) = |Bin_{i,f}(X) - Bin(X)|$ is an item's drift under fault pattern f and input i. To simplify our notation we say that $\Delta_{i,f} = \sum_{\forall X \in i} D_{i,f}(X)$.*

An item's drift is equal to the number of bins it has shifted away from its 'correct' bin during the RRS algorithm. Using drift we can obtain Lemma 1.

Lemma 1. *$\Delta_{i,f} \leq \epsilon|i|(|i| - 1) = \epsilon N(N - 1)$ when $|i| = N$.*

Proof. For an item to have a drift of $D_{i,f}(X)$, it must have experienced at least $D_{i,f}(X)$ failed comparisons during its pass of the RRS algorithm. Further, the number of failed comparisons of the RRS algorithm must be less than or equal to

$\epsilon|i|(|i|-1)$ since this is a worst-case analysis. Therefore, $\Delta_{i,f} \leq \epsilon N(N-1)$ when $|i| = N$.

□

Next we examine the *broken orderings*—the number of pairs of items that are misordered with respect to each other—in the output.

Definition 5. *Let $Pos_{i,f}(X)$ be the position of item X in the output of RRS run on input i and fault pattern f. Then:*

$$B_{i,f}(X,Y) = \begin{cases} 1 & X < Y \wedge Pos_{i,f}(X) > Pos_{i,f}(Y) \\ 0 & otherwise \end{cases}$$

is a broken ordering function of RRS*. To simplify our notation we say that:*

$$\beta_{i,f} = \sum_{X \in i} \sum_{Y \in i, Y > X} B_{i,f}(X,Y)$$

The $USFE$ is bound by a function that is proportional to the number of broken orderings, which we show in Lemma 2.

Lemma 2. $USFE(RRS_f(i)) \leq 2\beta_{i,f}$

Proof. When an item is displaced by L in either direction in the output it must participate in *at least L distinct* broken orderings. Additionally, at most 2 items can participate in each *distinct* broken ordering. Finally, $USFE$ is the sum of the displacement of every item in the output. Therefore, by the pigeonhole principle $USFE(RRS_f(i)) \leq 2\beta_{i,f}$.

□

The number of broken orderings, however, is directly related to the total drift of all items during a faulty run of the *RRS* algorithm. We show this in Lemma 3.

Lemma 3. $\beta_{i,f} \leq 2\Delta_{i,f}$.

Proof. Assign to each item X a purse containing $C_X = 2D_{i,f}(X)$ credits. X goes bankrupt if it ever has a negative number of credits. We proceed by showing an inductive procedure by which all items can pay for all broken orderings without any item going bankrupt.

Base Case: Let $X_1, X_2, ..., X_v$ all have drift D_M, the largest drift of all items. Then each of these items can participate in at most $2D_M$ broken orderings since no item has a drift greater than D_M (and thus cannot be further than D_M bins from its correct bin).

Therefore, each of the $X's$ can pay for all of the broken orderings they participate in with their own credits and none of them will go bankrupt.

Inductive Hypothesis: We will assume the following for this inductive proof:

(1) Items with drifts greater than D have paid for all the broken orderings they have participated in.

(2) None of the items with drift greater than D have gone bankrupt.

Notice that this true of the base case.

Inductive Step: Now, given the inductive hypothesis we will show that:

(1) Items with drifts greater than *or equal to* D have paid for all the broken orderings they have participated in.

(2) None of the items with drift greater than *or equal to* D have gone bankrupt.

Notice that if $X_1, X_2, ..., X_v$ have drifts of *exactly* D, then they do not have to pay for any broken orderings they may participate in with an item that has *more*

than D drift. Therefore, they are only responsible for broken orderings with items that have D or less drift. However, each of the $X's$ can only participate in at most $2D$ such broken orderings since there are only $2D$ items within D correct bins of each of these items.

Thus both (1) and (2) hold from the inductive hypothesis to the inductive step.

Conclusion:

Therefore, since (1) all broken orderings can be paid for by some item, (2) each item X has only $2D_{i,f}(X)$ credits to pay with, and (3) no item goes bankrupt, then $\beta_{i,f} \leq 2\Delta_{i,f}$.

□

By our three lemmas we can see that

$$\forall i \in I_N, f \in F_{N,\epsilon} USFE(RRS_f(i), i) \leq 4\epsilon N(N-1) \tag{6.6}$$

However, the normalization factor for $USFE$ is $\Theta(N^2)$ and therefore:

$$\forall i \in I_N, f \in F_{N,\epsilon} SFE(RRS_f(i), i) \leq \epsilon\Theta(C) \tag{6.7}$$

and since $\epsilon\Theta(C)$ goes to 0 as ϵ goes to 0, (RRS, SFE) is $WCSR$. □

In Chapter 7 I will explore the consequences of these theorems in greater detail and speculate on their meaning.

Worst-Case Scalable Robustness Results

	Merge	**Quick**	**Bubble**	**Round Robin**
Strict Correctness	X	X	X	X
Max Displacement	X	X	X	X
Spearman's Footrule	X	X	X	✓

Table 6.2: **Summary of *WCSR* Results.** Each (algorithm, context) pair marked with an X is provably *not* *WCSR*. The $\Theta(N^2)$ round robin sort algorithm, when paired with the Spearman's footrule error context, stands out as the only algorithm that is *WCSR*. Reprinted from [3] with permission.

Chapter 7

Discussion

7.1 Summary of Work

In this dissertation I explored empirical and theoretical behaviors of deterministic algorithms subject to failures that they cannot avoid. I explored the behavior of sorting algorithms subject to faulty comparisons, while my study on matrix multiplication algorithms made use of faulty bit checks, add statements, and scalar multiplication steps. I also built a general tool named *Criticality Explorer*, and I used it to evaluate the matrix multiplication algorithms. Finally, using my criticality observations on sorting algorithms as a guide, I developed a theoretical framework called scalable robustness to identify algorithms and error measures that degrade gracefully.

I also created a notion of “failure point” focused on measuring computational failures in both time and space. To support this concept I created method-level failure interfaces—generalizable descriptions of failure behavior for methods that allow for the study of *algorithmic* robustness to failures, separate from, and without detailed knowledge of, any specific hardware platform. I also bucketed method failures by *time*, allowing me to observe algorithmic responses to failures that occur at otherwise

seemingly equivalent operations throughout the algorithmic life cycle.

My measures of algorithmic robustness appropriated old domain-specific measures as output error measures. For sorting I employed presortedness, a traditional measure of problem difficulty used to evaluate list sort error. In matrix multiplication I used Infinity, Frobenius and log Frobenius error between matrices, measures that are used to evaluate matrix modeling error [105]. These error measures made it possible to observe complex algorithmic failure responses, by removing the barrier imposed by the strict correctness standard. Using the results of these analyses I was able to economize algorithmic resources in a way that decreased overall output error in a resource-constrained environment that didn't allow for the correction of all failures.

All together, from this work I draw the following conclusions concerning computational robustness.

7.2 Questions Answered

The major research questions of this dissertation were:

- How do measures of correctness hide or uncover interesting failure dynamics inside algorithms?
- How are an algorithm's failure dynamics related to the algorithm's known behavior?
- Is it possible to use an analysis of failure dynamics to improve algorithmic performance in resource-constrained environments?
- Which operations respond well to this method?

In the rest of this section I walk through my conclusions on each of these questions. I also present speculations on two additional questions that arose during the course of the study:

- How does numerical stability relate to matrix multiplication results?
- When Will Criticality Structures Scale Effectively?

7.2.1 Strict Correctness Hides Criticality Structure While Quantified Correctness Reveals It

Figure 4.1 (page 54) shows estimated criticalities under the strict correctness error measure. We can use this figure to understand some of the liabilities of all-or-none-correctness. While the figure does make clear that quick and merge sorts perform many fewer comparisons than bubble sort, little other structure is revealed. Despite averaging over random input permutations, the strict correctness criticality of each comparison is usually either 1 or 0: Any given comparison is either maximally critical or not at all critical. Given 0% background failures (*red* curve), for example, there will only be a single failure. For bubble sort, a failure is critical if that failure is in any of the last N comparisons (seen at about comparison 2600), but otherwise it's harmless. By contrast, with merge and quicksorts almost every comparison is critical. The last comparisons for quick and merge sorts show intermediate criticalities because, depending on the specific input permutation tested, the algorithm will sometimes finish before that comparison is reached, so a failure at that comparison index is sometimes harmless.

Given strict correctness, if the background failure rate is appreciably non-zero (e.g., 20%, *blue* curve) all comparisons became non-critical in all three algorithms: Since the output will never be strictly correct, the occurrence or absence of any one failure makes no difference.The one major exception to this strict correct rule is full

bubble sort which does not have every operation presenting full criticality at a 0% background failure rate, but only the last n operations. This is an interesting partial exception to Hypothesis 1.4.1 that suggests that for some inefficient algorithms the most interesting criticality behavior may be apparent under SCE.

However, if we look at Figures 4.2 (page 55), 4.3 (page 56), 4.4 (page 57), 5.1 (page 68), or 5.11 (page 79) we see many instances where error structures seem to be revealed by quantified correctness measures. The major exception is the brute force naive matrix multiplication which seems to present a very flat criticality measure for each operation even when measured by quantified correctness measures.

7.2.2 Criticality Structure Seems Related to Known Algorithmic Behaviors

In most of the algorithms I have presented, the observed criticality structures seem to be related to the known behaviors of the algorithms under consideration. We can see this in the each of the sorting algorithms: merge sort, quick sort, and bubble sort; as well as in the multiplication algorithms.

Merge Sort

In Figure 4.2 (page 55) we see average conditional Spearman's footrule error (positional error) and criticality for the merge sort algorithm. The criticality of a failure at a given comparison index—illustrated in the middle graph—is equal to the difference between the top and bottom lines in the first graph of Figure 4.2—the estimated error when the failures does occur less that when it doesn't.

We should note two striking aspects in the middle graph in Figure 4.2. First, the positional error measure seems to reveal a fractal criticality structure for the merge sort algorithm. In retrospect, at least, this makes sense given the depth-

first recursion used in my merge sort implementation. Comparisons at the deepest recursive levels—when two items are merged into a length 2 sublist—are also the most critical comparisons; the deeper "criticality valleys" reflect the larger merges.

Second, that recursive criticality structure is persistent across background failure rates. Even at a background failure rate of 20% we can still see four distinct 'bumps' in merge sort's criticality results. This implies that criticality structure is robust when the right algorithm and error measure are used. Note that the criticality falls off at larger background failure rates since criticality measures additional error due to a fault and at higher background failure rates so much damage has already been done to the output that it becomes difficult for failures to do even more damage.

Next, when comparing the middle graph of Figure 4.2 (page 55) to the bottom graph we see that max displacement error reveals a structure that is similar to that revealed by positional error. However, the structure of max displacement error collapses as the number of background errors increases. This seems to present max displacement error as an interesting intermediary between strict correctness error and Spearman's footrule error, a conclusion which is also supported by Theorem 8 (page 110) in Chapter 6.

Quicksort

If we move beyond merge sort to look at quicksort paired with quantified correctness measures, as in Figure 4.3 (page 56), we see similar structured results as in merge sort: With Spearman's footrule error criticality, the first $N = 52$ comparisons have much greater criticalities than all other comparisons in the sort. This occurs because the first N comparisons of quicksort are responsible for sorting the list into a 'top' half and 'bottom' half with all items less than the pivot in the bottom half and all items greater than the pivot in the top half. A failed comparison in the first N comparisons leads to the miscompared item being placed, on average, about $N/2$

away from its correct position.

On the other hand, for inversions error criticality, the first N comparisons have *lower* criticalities than all other comparisons. I suspect this is because items misplaced in either the top or bottom half of the list will tend to move toward the center where the halves meet, so even many failures in the first N comparisons will tend to add only a single inversion to the error measure.

Though Spearman's footrule error and inversions error, like all the list error measures explored in this dissertation, agree on the meaning of 'correctly sorted', they measure different list properties, and their criticality structures are sometimes quite different. While we may hope to find general principles, it is important to understand that a wise choice of error measure requires not only sensitivity to failures, but also to the needs of a computation's end-user.

In Figure 4.3 (page 56) the spikes at the beginning of the first two passes through the list seem anomalous and at this time I do not have an explanation for their existence.

Bubble Sort

Next, consider the bubble sort results in Figure 4.4 (page 57). Bubble sort's $O(N^2)$ comparisons give it a great deal of redundancy, so the criticalities in Figure 4.4 are much smaller than in, say, Figure 4.3 (page 56). In addition, the details of its criticality structures emerge at high background failure rates. It is unsurprising that bubble sort's last N comparisons are its most critical as any failed comparison in the last N operations cannot be corrected, but I hadn't anticipated the small but distinct length N periodic structure throughout bubble sort's execution, indicating increased criticality in the last half of each pass through the list.

All three sorting algorithms displayed structures related to the input size of $N =$

52: First N criticality in quicksort, last N criticality in merge sort, and a period N variation in bubble sort.

Scalar Multiplication

In Figure 5.1 we see how operations in scalar multiplication algorithms relate to algorithm output. For naive multiplication, criticalities appear to grow logarithmically as the algorithm proceeds through one of the multiplicands, summing the other multiplicand depending on the results of each check operation. Alternatively, for Karatsuba multiply we can observe a constellation of three maxima in the graph of operation criticality at 0, 100 and 250 `check` or `add` operations at multiplication scale $N = 500$. Similarly, between the maxima we seem to observe similar, fractal maxima at smaller scales. This makes sense for an algorithm that works by recursively splitting each multiplication operation into three sub-operations. There is a replication of patterns at size 100 and size 500; this suggests that more investigation is warranted.

In Figures 5.8 (page 75), 5.9 (page 76), and 5.10 (page 77) I show failure shaped error results against i.i.d. failure model outcomes for each failure interface and error measure at input size 100. Failure shaping cuts the average absolute logarithmic failure by about 30. It cuts the absolute value and absolute percent value by between 10% and 90%. Further, the Karatsuba algorithm benefits more from failure shaping than naive multiply. A possible reason for the discrepancy between the absolute value and its logarithmic version is due to the high standard error of the sampling procedure using this time based criticality technique. Most multiplication runs for both naive and Karatsuba multiply had failure-shaped error outputs that were many orders of magnitude lower than non-failure-shaped error outputs. However, a small number did not have such great improvements and it is these instances that dominate the absolute value measurement while having very little impact on the log absolute value

error measure. In some sense, these instances are not successfully failure shaped.

Matrix Multiplication

Figure 5.11 (page 79) presents results for our two matrix multiplication algorithms. For naive matrix multiply we don't see much criticality structure—criticality appears flat. However, for Strassen's algorithm we do see spikes that appear about 2/7, 3/7 and 5/7 of the way through the algorithm. This makes sense if we consider that the algorithm works by splitting the input matrices into 8 parts and recursing 7 times to obtain its speed up, though the reason for the spikes on these specific ranges of 1/7 of program run time is unclear. We also observe that this structure holds not only for scalar multiplication proxy operations, but it also holds across multiple input sizes, suggesting that criticality at small scales may be useful in evaluating criticality at larger scales, at least for Strassen's algorithm. In naive matrix multiply, each scalar multiplication has an equal impact on the output, and its criticality structure is correspondingly flat.

7.2.3 Algorithmic Performance in Resource-Constrained Environments Is Improved Through Failure Dynamic Analysis

Reviewing Figures 5.8 (page 75), 5.9 (page 76), 5.10 (page 77), 5.25 (page 93), 5.26 (page 94), and 5.27 (page 95) we can see that it is indeed possible to decrease total output error in resource-constrained environments by using criticality assessment results. However, two conditions must be met for this to work. First, the failure shapes of the algorithms must contain high *leverage*—a measure of the difference between critical and non-critical operations in a program. Second, the representation of failures needs to be *compact*—short compared to the run time of the algorithm—

and present us with low *variance*.

The Importance of Leverage

I call the ratio of the average criticalities of a program's important and unimportant operations its *leverage*. I use the median failure point criticality as the dividing line between important and unimportant operations, however other policies may perform better for other algorithms and economizations. Leverage is thus a rough-and-ready hint of the improvements possible via my implementation of failure shaping.

In Figure 7.1 the leverage of both naive and Strassen's matrix multiply using the *Frobenius norm* are presented. Strassen's matrix multiply shows a higher leverage than naive matrix multiplication, which also outperforms Strassen's matrix multiplication on baseline i.i.d. failure tests. However, because Strassen's algorithm has higher leverage, it also responds better to the failure shaping procedure.

The evidence is more murky with the scalar multiplication algorithms. This may be due to the the large constant term in Karatsuba's growth rate. In the limit, Karatsuba is *more efficient* than naive scalar multiply, but *at my experiment scales* it was less efficient. Larger experiments are necessary to resolve the question of leverage in scalar multiplication algorithms. However, Karatsuba's leverage does seem to pick up as we move from 200 to 500 bits, indicating that at larger scales Karatsuba's leverage may be higher than naive multiplication's.

Overall, economic failure shaping seems best suited to computations that

1. perform multiple fallible steps,
2. each of which has a definable cost,
3. at definable failure rates, with
4. high leverage, and

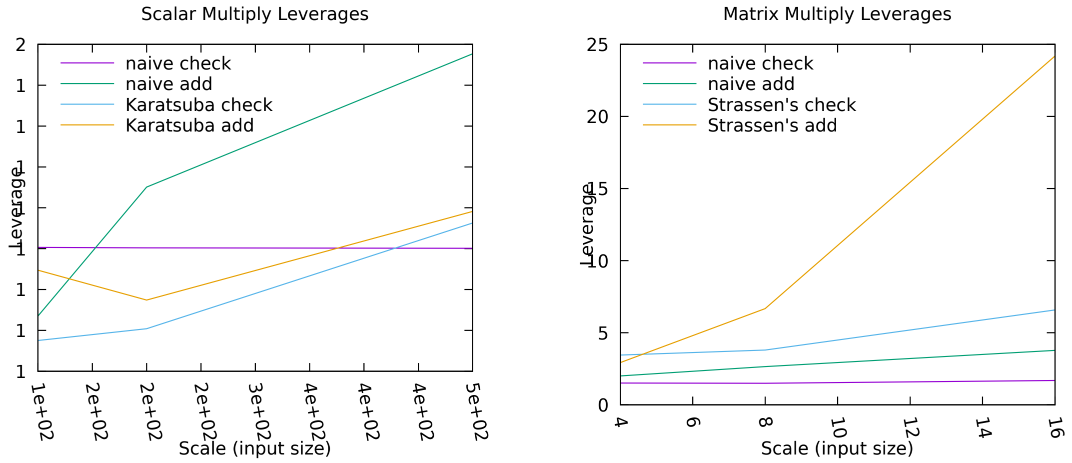


Figure 7.1: **Selected Leverage Results.** *Log absolute value* leverages on both naive and Karatsuba scalar multiplication algorithms are presented on the left. *Frobenius norm* leverage results on the matrix multiply algorithms are presented on the right. Note that Karatsuba leverage on `check` and `add` operations seems to dip before picking up and growing above naive scalar multiply's `check` leverage at scale 500. Strassen's algorithm leverage on all operations grows faster than naive matrix multiply's leverage on any operation. Note that graphs have different x and y axes. See text for details. Reprinted from [1].

5. limited overall resources.

Although satisfying most of these conditions is a matter of framing the question, condition 4 depends on the underlying algorithms. However, I do find that *high efficiency* algorithms often have high leverage. This is true in sorting: The more efficient algorithms have much higher max criticalities than the $O(N^2)$ full bubble sort algorithm. Intuitively, this makes sense, as efficiency often depends on making high-impact decisions about the output based on examining as little data as possible at the decision point.

To satisfy the other four conditions, we seek out the operations that are most heavily impacted when increasing algorithmic efficiency. So, for example, compar-

isons in sorting, and scalar multiplications in matrix multiplication are both economized as algorithms become more efficient. In addition, looking beyond the algorithms explored in this work, graph algorithms often economize on the number of vertex or edge traversals necessary to produce a correct output.

Assessment Compactness and Variance

Failure points with high *variance* occur when many different kinds of execution paths overlap, some important and some unimportant while *compact* descriptions grow no more quickly than the algorithm they are describing. There is a necessary tension between these two concepts.

For my study, failure points were bucketed by method name and invocation count. An alternative bucketing might be to evaluate every possible execution path on every possible input for its error outcome. Such a bucketing would, undoubtedly, have low variance, however, it would not be compact. By contrast, a bucketing that lumps all errors together would be very compact, but would have high variance.

My failure interfaces produce compact space-time descriptions. However, variances on the scalar multiplication algorithms were high enough to make failure shaping difficult. Too many high variance and high impact `add` operations fail for the full expected usefulness of failure shaping to appear anywhere on the scalar multiplication algorithms except on the *log absolute value* score. That score calculates logarithms before performing averages. Thus unusual experiment runs with scores orders of magnitude greater or lower than the average experiment, have a lower impact there than on the *absolute value* error measure.

7.2.4 Bulk Program Operations Make Good Choice for Failure Shaping

The operations I used in this work are often used to determine lower bounds on algorithmic efficiency—comparisons in sorting and check and add operations in multiplication. These operations often make up the bulk of an algorithm's run time and are also often somewhat parallelizable. So far operations from this category appear to be useful for failure shaping, excluding naive algorithms like bubble sort and naive matrix multiplication. This hypothesis, however, could use a greater degree of support beyond these three problem domains.

I speculate that naive algorithms in a given problem domain contain the best operations for failure shaping. Scalably robust algorithms only experience a small amount of output error for every given internal failure. When a naive algorithm is paired with an input generator, operation failure mode, and error measure that is provably *scalably robust*, then we are likely to observe criticality behavior that can be successfully failure shaped on more efficient algorithms in the same problem domain. However, the replication of operations limits this hypothesis, as articulated by Theorem 1 where arbitrary $lgf(N)$ operation replication implies that even strict correctness is always scalably robust. Another limit on this hypothesis is the lack of a formal definition for 'naive' algorithms in a problem domain.

7.2.5 Numerical Stability

Beyond leverage, the criticality results for matrix multiplication may also have an interesting connection to questions of numerical stability. Strassen's algorithm, also called "fast matrix multiply", is less numerically stable than naive matrix multiply [106]. As a result, it is less likely to be chosen for many computing tasks. I hypothesize that efficient matrix multiplication algorithms are unstable *because* their

internal leverage scales up the effects of small rounding errors in return for faster run times. In the case of Strassen's algorithm, this appears to occur during the 2nd, 3rd, and 5th recursive steps.

7.2.6 When Will Criticality Structures Scale Effectively?

In both matrix and scalar multiplication we see the similar criticality structures at multiple scales. In Karatsuba multiplication the criticality graphs present a three maxima pattern at multiple scales while in Strassen's matrix multiplication there are criticality spikes at failures points 2/7, 3/7, and 5/7 of the way through the algorithm at multiple input scales. These similarities suggest that small scale criticality experimental results may be useful in evaluating larger scale programs.

7.3 Limitations of the Study

The important limitations of this study include:

- **The focus on a limited number of fallible operations for each problem domain.**

  The choice to limit the scale of this dissertation to the constrained reliability allocation problem—which focuses on operations that only produce output errors, not program crashes or infinite loops—led to the focus on a limited number of operations per algorithm. However, even though the *type* of operations that were studied were limited, I focused on operations that do a great percentage of the work of a program—those operations that tend to be used to analyze program efficiency. Further, as discussed in Chapters 1 and 2, there is reason to believe that in the future it will be possible to constrain resources

at specific failure points and 'green/white' computing imagines a world where only some program operations are allowed to fail [26].

- **The focus on i.i.d. failure distributions at the cost of exploring more complex models that make use of coordinated failures.**

  As stated in Section 2.7, one of the ideas of this dissertation is that coordination at one level of computation leads to randomization at another level. Rather than focusing on coordinated failures at the hardware level, I have instead focused on i.i.d. failures at the program level that can only be *created by* coordinated failures beneath the program level. However, this move deals with coordinated hardware failures by assuming they will have a particular effect rather than by *showing* they have that effect. Future studies should examine the actual impact of coordinated hardware failures on programs at run time. I present ideas for future work in this direction in the next section.

- **The choice of a simple error economization.**

  The error economization I've used in this thesis is very simple, each percentage point decrease in epsilon for one operation has the exact same costs as a percent point decrease in epsilon anywhere else. Operations can freely substitute their resources for another operation's resources. This approximation will only be good for some real world situations. More complicated error economizations that take into account the downstream costs of failed chip timings or that more closely track the interaction between operation error and energy are necessary.

- **The choice of a simple failure *redistribution policy.***

  In this dissertation I have chosen to redistribute failure resources from operations with criticalities beneath the median to those above the median. This is a very simple redistribution scheme, however, and may not be the most efficient policy for failure redistribution. Specifically, I ignore statistical and continuous

methods of failure redistribution that may produce better outcomes. I address this weakness in the next section in future work.

- **The use of simple failure modes that are not necessarily representative of the behavior of real world SDCs.**

  In this thesis I made use of handcrafted 'failure modes' that will not always track the behavior of real world SDCs. This included flipping the output of a comparison operation and a check operation, and randomly flipping a single bit in the output of an add or multiplication operation. In future work I will propose a path of study to address this limitation of the work.

- **Limited set of problems explored.**

  This study is limited to the *sorting*, *scalar multiplication*, and *matrix multiplication* problem domains. Any empirical study will be limited to some problem areas, but empirical evidence from others would still be welcome. In future work I will present other potential problem domains.

- **A lack of proofs concerning some (sorting algorithm, error measure) tuples in Chapter 6 and about programs outside of the sorting problem domain.**

  The proofs in Chapter 6 are interesting since they show that looser definitions of 'correct' lead to a greater tendency to achieve average case scalable robustness. However, the relationship between scalable robustness and program efficiency within the program-error tuples that seem closest to the transition boundary between scalably robust and not scalably robust is still poorly understood. Further, outside of a few general statements about the relationship between worst and average case scalable robustness, and the use of modular redundancy in scalable robustness, this work is limited to sorting. I address both of these weaknesses in future work.

- **The limited scale of the empirical study.**

Due to the run time of the criticality assessment procedure, as presented in this thesis, it was not possible to collect criticality data concerning very large program instances. However, in future work I will present some possible paths forward on this question.

7.4 Future Work

I find seven major possible research paths moving forward: Expanded empirical studies that seek criticality assessments and error economizations on unaddressed domains, scaling of the criticality assessment technique through search-based techniques, scaling of the criticality assessment technique through machine learning attribution models, studies of hardware behavior that can be used to inform operation failure modes, spatial graphs for coordinated failure models, the development of stochastic and continuous failure redistribution policies, and theoretical work that moves beyond the sorting domain.

7.4.1 Expanded Empirical Studies

The empirical studies presented in this dissertation include the sorting, scalar multiplication, and matrix multiplication problem domains. While the techniques presented here have found interesting results in these domains, this is still a limited data set. Empirical studies of criticality should be expanded into new domains such as compression, graph problems such as max flow or min cut, and database operations.

7.4.2 Scaling Through Search

Failure shaping currently requires significant human labor. Making it more efficient will involve leveraging multiple strategies. One strategy includes a library of

standardized input types, error measures, and failure interfaces that can be used to produce annotated programs in the future.

Such a strategy, though, will tend to increase the computational resources required by the method. The programs I have presented are sufficiently limited that they can be characterized relatively cheaply, but larger programs will be more difficult to explore. My observation of apparently similar failure shapes at multiple scales (see Figure 5.11—page 79) suggests one strategy could be to scale the criticality assessment by run time in some programs. However, we will also need more sophisticated search methods—ones capable of performing significant *generalization* across failure points, rather than gathering fully-independent statistics as I have here.

Genetic algorithms, genetic programming, and other adaptive search procedures are often employed to search combinatoric spaces, as in [107–109]. A common problem in this space is the flag variable problem [110]. In [111], the authors note that GAs work best in search spaces that avoid these "needle-in-the-haystack" spikes. My use of continuous error measures compared to typical all-or-none test failures may help produce such 'softened' search space gradients, as medium-criticality operations tend to cluster around spikes both here and in other algorithms we have explored [2].

This is a new area, but a relatively sparse set of data-points plus a suitable heuristic search procedure may allow us to build imperfect but high-quality *criticality estimators* for the failures of much larger pieces of software than are reachable via Monte Carlo search alone.

7.4.3 Machine Learning and Attribution

Another possible path forward for failure shaping involves the use of deep learning attribution models [112] paired with a shallow error study at a some low ϵ. Deep learning is a machine learning technique that learns layers of mappings from inputs,

represented as vectors of floating point numbers, to outputs represented the same way. These mappings are represented as a series of non-linear functions from vectors to vectors, where each dimension in an output vector is a linear combination of dimensions on the input, followed by non-linear activation functions such as tanh, relu, or a logistic function.

The original inspiration for deep learning are neural networks found in biological systems, where a combined linear step and non-linear activation function is often conceived as being similar to a single neuron. A good mapping over the entire network is found through a process of gradient descent, where inputs are mapped through the network, a potentially erroneous output is calculated, the distance between the erroneous output and the correct results is then evaluated, and then the internal weights of each neuron are modified by a derivative that is roughly equivalent to each neuron's contribution to the final erroneous output value.

One of the difficulties with a deep learning model, however, involves attributing the model's reasoning about output values to specific input dimensions. However, a method called *integrated gradients* has recently been proven to have a number of useful properties for attribution assignment and has been used to find attributions that seem reasonable in a number of deep learning models [113].

It may be possible to use a method like integrated gradients to perform criticality evaluations at large scale. If we treat a failure pattern as the input and the final error as the output we may then be able to train a deep learning model on a constant number of program instances with some reasonably small i.i.d. background failure rate. This represents a speed up of the method as it will no longer be necessary to perform a large number of simulations for each individual computational step. After the model is trained, we can then attribute the final output error results to each of the operations, evaluating which operations are most important in determining output error.

7.4.4 Hardware Failure Studies

A comparative study that looks at the effect of hardware failures introduced by lower energy usage could be useful to fill the gaps in our knowledge concerning the failure modes of deterministic operations used by algorithms. These studies could be enhanced by looking at major virtualization techniques, such at the JVM, that may introduce even further changes in the distribution of likely failure modes for a give operation. Reasonable failure modes may be composed by combining the likely behavior of a given hardware system with the likely behavior of the software stack sitting above hardware.

7.4.5 Spatial Graphs for Coordinated Failures

In the current study only i.i.d. and worst case failure models are presented. In the future it would be useful to see models that increase or decrease individual operation failure rates based on the failure of nearby operations as in [78]. In that work, comparison failures were coordinated if they were close to each other in time. Similar models that make use of operation neighbor graphs to coordinate failures may be used to surpass the i.i.d. paradigm. Such coordination graphs may place operations close to each other if they have similar semantics, operate on the same CPU, or are close to each other in time.

7.4.6 Stochastic and Continuous Failure Redistribution Policies

In this work I used a simplified failure redistribution policy that redirected resources from operations with criticalities beneath the median to those with criticalities above the median. This policy improved algorithmic robustness, however there is no *a pri-*

ori reason to think this is the most efficient policy for failure shaping. Alternative policies might include redirecting resources stochastically using continuous probability distributions based on operation criticality and other information as opposed to the deterministic decision process used in this work.

7.4.7 Scalable Robustness Beyond Sorting

In this dissertation I have presented a generalized concept of scalable robustness, paired with proofs that expose the behavior of a number of sorting algorithms considered in the light of the scalable robustness paradigm. These proofs suggest an interesting relationship between error measure, algorithm efficiency, and robustness. However, further research paths may be opened by theoretical work on the scalable robustness of programs outside the sorting domain.

7.5 Significance

Here I summarize the contributions of this work:

1. This work helps uncover failure dynamics in complex pieces of code by outlining a method for their consistent discovery. It does this by pointing out the need for error measures that evaluate output quality instead of hard correctness measures. Further, it begins the process of building a body of failure modes for the purposes of examining the dynamics of failures inside of computations.

2. This work aids in the armoring of computations against internal failures by providing a measure of each operational instance's criticality. Given this information, it is a small step to see that computational architects can choose to armor only those operational instances with a great deal of criticality, thus reducing the cost of building robust computations. This allows computational ar-

chitects to make choices between heavily armored, robust computations which require a great deal of resources and less armored highly efficient computations that pass a greater number of failures through to their outputs. This extends the green/white approach to energy saving through approximate computing, allowing for subtle gradations of failure-approximation that depend on specific algorithmic behavior.

3. This work provides researchers with a new view on the internal workings of algorithms. This view may provide computer scientists with new methods for classifying and identifying and teaching algorithms.

4. In modern computer systems, built on large deterministic hardware, functional modules are often designed, deployed, and composed with virtually no knowledge of the overall system behavior. This dissertation's fourth contribution is the notion of *method level failure interfaces* that allow the study of approximate computations separate from their underlying hardware stack, either freeing us from the need for specialized hardware knowledge, or at least allowing us to compile that knowledge in a separate process.

5. Unlike traditional approaches that use fault tolerance to reduce or eliminate failures, I have a 'hardware reliability budget' that treats failure rates as an independent variable: *guaranteeing that failures cannot be avoided*. This dissertation moves beyond fault-tolerance/intolerance by dealing with reliability resource allocation when those resources are not sufficient for strictly correct algorithmic results.

6. The theoretical views provided in this work are the first treatment, to my knowledge, that considers the robustness of an algorithm as input sizes grow infinitely while error rates approach zero. One major advance introduced by this perspective is Theorem 1 (page 1) which shows a bound on the *maximum* redundancy needed to ensure the correct operation of a program given an i.i.d.

failure model. The proofs relating sorting algorithm efficiency, error measure, and scalable robustness are also suggestive of a previously unobserved theory of failure dynamics.

7.6 Conclusion

The work presented in this dissertation is both incomplete and exciting. The relationships I have observed—between program operation failures and program output error—are related to both known, and newly discovered, algorithmic behaviors and properties. This is especially true for program efficiency, which impacts whether particular algorithms can be considered robust at all. This work also shows that these relationships can be leveraged to improve algorithmic behavior in the face of failure-prone hardware or even in the face of other sources of error, such as rounding. The possibilities seem boundless when we peer behind the barrier imposed by strict correctness.

Appendices

Appendix A

Permission to Reproduce Previously Published Content

The University of New Mexico requires that previously published works included in this dissertation be licensed by their respective copyright holders. Copies of those licenses must be included in the dissertation. If the material was published under an open access license the guidelines for use must be included in lieu of a license. Accordingly this appendix includes the following documents:

- Instructions for reproduction of IEEE publications in a dissertation. Some contents of Chapters 4 and 6 previously appeared in the proceedings of *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks* and *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop*, respectively, both published by the IEEE.

- A license to reproduce material in Chapters 3 and 5 both published in the proceedings of the *International Symposium on Search Based Software Engineering* which for which Springer Nature is the copyright holder.

A.1 IEEE Reproduction Instructions

The IEEE does not require individuals working on a thesis to obtain a formal reuse license, however, you may print out this statement to be used as a permission grant:

Requirements to be followed when using any portion (e.g., figure, graph, table, or textual material) of an IEEE copyrighted paper in a thesis:

1. In the case of textual material (e.g., using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line © 2011 IEEE.
2. In the case of illustrations or tabular material, we require that the copyright line © [Year of original publication] IEEE appear prominently with each reprinted figure and/or table.
3. If a substantial portion of the original paper is to be used, and if you are not the senior author, also obtain the senior author's approval.

Requirements to be followed when using an entire IEEE copyrighted paper in a thesis:

1. The following IEEE copyright/ credit notice should be placed prominently in the references: © [year of original publication] IEEE. Reprinted, with permission, from [author names, paper title, IEEE publication title, and month/year of publication]
2. Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis on-line.
3. In placing the thesis on the author's university website, please display the following message in a prominent place on the website: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does

> not endorse any of [university/educational entity's name goes here]'s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to `http://www.ieee.org/publications_standards/publications/rights/rights_link.html` to learn how to obtain a License from RightsLink.

If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

A.2 Springer Nature License Terms and Conditions

Nov 05, 2018

This Agreement between Mr. Thomas Jones ("You") and Springer Nature ("Springer Nature") consists of your license details and the terms and conditions provided by Springer Nature and Copyright Clearance Center.

License Number	4462630773902
License date	Nov 05, 2018
Licensed Content Publisher	Springer Nature
Licensed Content Publication	Springer eBook
Licensed Content Title	Damage Reduction via White-Box Failure Shaping
Licensed Content Author	Thomas B. Jones, David H. Ackley
Licensed Content Date	Jan 1, 2018
Type of Use	Thesis/Dissertation
Requestor type	academic/university or research institute
Format	print and electronic
Portion	full article/chapter
Will you be translating?	no
Circulation/distribution	$> 50,000$
Author of this Springer Nature content	yes
Title	Damage Reduction via White-Box Failure Shaping
Institution name	University of New Mexico
Expected presentation date	Nov 2018

Requestor Location	Mr. Thomas Jones 1305 NE 43rd ST Apt 809 SEATTLE, WA 98105 United States Attn: Mr. Thomas Jones Billing Type Invoice Billing Address Mr. Thomas Jones 1305 NE 43rd ST Apt 809 SEATTLE, WA 98105 United States Attn: Mr. Thomas Jones
Total	0.00 USD
Terms and Conditions	

Springer Nature Terms and Conditions for RightsLink Permissions

Springer Nature Customer Service Centre GmbH (the Licensor) hereby grants you a non-exclusive, world-wide licence to reproduce the material and for the purpose and requirements specified in the attached copy of your order form, and for no other use, subject to the conditions below:

1. The Licensor warrants that it has, to the best of its knowledge, the rights to license reuse of this material. However, you should ensure that the material you are requesting is original to the Licensor and does not carry the copyright of another entity (as credited in the published version).

   If the credit line on any part of the material you have requested indicates that

it was reprinted or adapted with permission from another source, then you should also seek permission from that source to reuse the material.

2. Where **print only** permission has been granted for a fee, separate permission must be obtained for any additional electronic re-use.

3. Permission granted **free of charge** for material in print is also usually granted for any electronic version of that work, provided that the material is incidental to your work as a whole and that the electronic version is essentially equivalent to, or substitutes for, the print version.

4. A licence for 'post on a website' is valid for 12 months from the licence date. This licence does not cover use of full text articles on websites.

5. Where **'reuse in a dissertation/thesis'** has been selected the following terms apply: Print rights of the final author's accepted manuscript (for clarity, NOT the published version) for up to 100 copies, electronic rights for use only on a personal website or institutional repository as defined by the Sherpa guideline (`www.sherpa.ac.uk/romeo/`).

6. Permission granted for books and journals is granted for the lifetime of the first edition and does not apply to second and subsequent editions (except where the first edition permission was granted free of charge or for signatories to the STM Permissions Guidelines `http://www.stm-assoc.org/copyright-legal-affairs/permissions/permissions-guidelines/`), and does not apply for editions in other languages unless additional translation rights have been granted separately in the licence.

7. Rights for additional components such as custom editions and derivatives require additional permission and may be subject to an additional fee. Please

apply to `Journalpermissions@springernature.com/bookpermissions@springernature.com` for these rights.

8. The Licensor's permission must be acknowledged next to the licensed material in print. In electronic form, this acknowledgement must be visible at the same time as the figures/tables/illustrations or abstract, and must be hyperlinked to the journal/book's homepage. Our required acknowledgement format is in the Appendix below.

9. Use of the material for incidental promotional use, minor editing privileges (this does not include cropping, adapting, omitting material or any other changes that affect the meaning, intention or moral rights of the author) and copies for the disabled are permitted under this licence.

10. Minor adaptations of single figures (changes of format, colour and style) do not require the Licensor's approval. However, the adaptation should be credited as shown in Appendix below.

Appendix - Acknowledgements:

For Journal Content:
Reprinted by permission from [**the Licensor**]: [**Journal Publisher** (e.g. Nature/Springer/Palgrave)] [**JOURNAL NAME**] [**REFERENCE CITATION**(Article name, Author(s) Name), [**COPYRIGHT**] (year of publication)

For **Advance Online Publication papers:**
Reprinted by permission from [**the Licensor**]: [**Journal Publisher** (e.g. Nature/Springer/Palgrave)] [**JOURNAL NAME**] [**REFERENCE CITATION**(Article name, Author(s) Name), [**COPYRIGHT**] (year of publication), advance online publication, day month year (doi: 10.1038/sj.[JOURNAL

ACRONYM].)

For Adaptations/Translations:
Adapted/Translated by permission from [**the Licensor**]: [Journal Publisher (e.g. Nature/Springer/Palgrave)] [**JOURNAL NAME**] [**REFERENCE CITATION**(Article name, Author(s) Name), [**COPYRIGHT**] (year of publication)

Note: For any republication from the British Journal of Cancer, the following credit line style applies:

Reprinted/adapted/translated by permission from [**the Licensor**]: on behalf of Cancer Research UK: : [Journal Publisher
(e.g. Nature/Springer/Palgrave)] [**JOURNAL NAME**] [**REFERENCE CITATION** (Article name, Author(s) Name), [**COPYRIGHT**] (year of publication)

For **Advance Online Publication papers:**
Reprinted by permission from The [**the Licensor**]: on behalf of Cancer Research UK: [Journal Publisher
(e.g. Nature/Springer/Palgrave)] [**JOURNAL NAME**] [**REFERENCE CITATION** (Article name, Author(s) Name), [**COPYRIGHT**] (year of publication), advance online publication, day month year (doi: 10.1038/sj.[JOURNAL ACRONYM])

For Book content:
Reprinted/adapted by permission from [**the Licensor**]: [Book Publisher (e.g. Palgrave Macmillan, Springer etc) [**Book Title**] by [**Book author**(s)]

[**COPYRIGHT**] (year of publication)

Other Conditions:

Version 1.1

Questions? `customercare@copyright.com` **or +1-855-239-3415 (toll free in the US) or +1-978-646-2777.**

References

[1] T. B. Jones and D. H. Ackley, "Damage reduction via white-box failure shaping," in *International Symposium on Search Based Software Engineering*, pp. 213–228, Springer, 2018.

[2] T. B. Jones and D. H. Ackley, "Comparison criticality in sorting algorithms," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 726–731, IEEE, 2014.

[3] T. B. Jones and D. H. Ackley, "Scalable robustness," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop*, pp. 31–38, IEEE, 2016.

[4] D. A. Oliveira, C. B. Lunardi, L. L. Pilla, P. Rech, P. O. Navaux, and L. Carro, "Radiation sensitivity of high performance computing applications on Kepler-based GPGPUs," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 732–737, IEEE, 2014.

[5] V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Analysis and characterization of inherent application resilience for approximate computing," in *Proceedings of the 50th Annual Design Automation Conference*, p. 113, ACM, 2013.

[6] J. von Neumann, "The general and logical theory of automata," in *Cerebral Mechanisms in Behaviour: the Hixon Symposium (1948)* (L. A. Jeffress, ed.), pp. 15–19, Wiley, 1951. Also appears as pages 302–306 in A.H. Taub, editor, *John von Neumann Collected Works: Volume V – Design of Computers, Theory of Automata and Numerical Analysis*, Pergamon Press, 1963.

[7] F. R. Shapiro, *The Yale book of quotations*. Yale Univ Pr, 2006.

[8] Computerworld, "Moth in the machine: Debugging the origins of 'bug'," Sep 2011.

[9] J. Von Neumann, “Probabilistic logics and the synthesis of reliable organisms from unreliable components,” *Automata studies*, vol. 34, pp. 43–98, 1956.

[10] A. Avižienis, “Fault-tolerance and fault-intolerance: Complementary approaches to reliable computing,” *SIGPLAN Not.*, vol. 10, pp. 458–464, Apr. 1975.

[11] A. Grnarov, J. Arlat, and A. Avizienis, “On the performance of software fault tolerance strategies,” in *Proc. 10th IEEE Int. Symp. Fault-Tolerant Computing*, pp. 251–253, Citeseer, 1980.

[12] S. K. Reinhardt and S. S. Mukherjee, “Transient fault detection via simultaneous multithreading,” in *ACM SIGARCH Computer Architecture News*, vol. 28, pp. 25–36, ACM, 2000.

[13] T. Vijaykumar, I. Pomeranz, and K. Cheng, “Transient-fault recovery using simultaneous multithreading,” *ACM SIGARCH Computer Architecture News*, vol. 30, no. 2, pp. 87–98, 2002.

[14] M. Gomaa, C. Scarbrough, T. Vijaykumar, and I. Pomeranz, “Transient-fault recovery for chip multiprocessors,” in *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, pp. 98–109, IEEE, 2003.

[15] J. Ray, J. C. Hoe, and B. Falsafi, “Dual use of superscalar datapath for transient-fault detection and recovery,” in *Microarchitecture, 2001. MICRO-34. Proceedings. 34th ACM/IEEE International Symposium on*, pp. 214–224, IEEE, 2001.

[16] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, “SWIFT: Software implemented fault tolerance,” in *Proceedings of the international symposium on Code generation and optimization*, pp. 243–254, IEEE Computer Society, 2005.

[17] P. P. Shirvani, N. R. Saxena, and E. J. McCluskey, “Software-implemented EDAC protection against SEUs,” *Reliability, IEEE Transactions on*, vol. 49, no. 3, pp. 273–284, 2000.

[18] N. Oh, P. P. Shirvani, and E. J. McCluskey, “Error detection by duplicated instructions in super-scalar processors,” *Reliability, IEEE Transactions on*, vol. 51, no. 1, pp. 63–75, 2002.

[19] R. Venkatasubramanian, J. P. Hayes, and B. T. Murray, “Low-cost on-line fault detection using control flow assertions,” in *On-Line Testing Symposium, 2003. IOLTS 2003. 9th IEEE*, pp. 137–143, IEEE, 2003.

[20] F. Cappello, A. Geist, B. Gropp, L. V. Kal, B. Kramer, and M. Snir, "Toward exascale resilience.," *IJHPCA*, vol. 23, no. 4, pp. 374–388, 2009.

[21] H. McCracken, "The year that software bugs ate the world," Dec 2017.

[22] J. Bent, G. Grider, B. Kettering, A. Manzanares, M. McClelland, A. Torres, and A. Torrez, "Storage challenges at Los Alamos national lab," in *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pp. 1–5, IEEE, 2012.

[23] K. Ferreira, J. Stearley, J. H. Laros III, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, and D. Arnold, "Evaluating the viability of process replication reliability for exascale systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, p. 44, ACM, 2011.

[24] S. Filiposka, A. Mishev, and C. Juiz, "Current prospects towards energy-efficient top HPC systems," *Computer Science and Information Systems*, vol. 13, no. 1, pp. 151–171, 2016.

[25] D. Mohapatra, V. K. Chippa, A. Raghunathan, and K. Roy, "Design of voltage-scalable meta-functions for approximate computing," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*, pp. 1–6, IEEE, 2011.

[26] B. Huang, R. Sass, N. Debardeleben, and S. Blanchard, "Harnessing unreliable cores in heterogeneous architecture: The PyDac programming model and runtime," in *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, pp. 744–749, IEEE, 2014.

[27] D. H. Ackley and E. S. Ackley, "Artificial life programming in the robust-first attractor," in *Proc. of the European Conference on Artificial Life (ECAL)*, (York, United Kingdom), July 2015.

[28] D. H. Ackley, "Indefinite scalability for living computation," in *Proc. of the Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[29] R. Akram, M. M. U. Alam, and A. Muzahid, "Approximate lock: Trading off accuracy for performance by skipping critical sections," in *2016 IEEE 27th International Symposium onSoftware Reliability Engineering (ISSRE)*, pp. 253–263, IEEE, 2016.

[30] V. V. Vazirani, *Approximation algorithms*. Springer Science & Business Media, 2001.

[31] J. Han and M. Orshansky, "Approximate computing: An emerging paradigm for energy-efficient design," in *2013 18th IEEE European Test Symposium (ETS)*, pp. 1–6, IEEE, 2013.

[32] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, "Flikker: saving dram refresh-power through critical data partitioning," *ACM SIGPLAN Notices*, vol. 47, no. 4, pp. 213–224, 2012.

[33] A. Raha and V. Raghunathan, "Towards full-system energy-accuracy tradeoffs: A case study of an approximate smart camera system," in *Proceedings of the 54th Annual Design Automation Conference 2017*, p. 74, ACM, 2017.

[34] V. Estivill-Castro and D. Wood, "A survey of adaptive sorting algorithms," *ACM Computing Surveys (CSUR)*, vol. 24, no. 4, pp. 441–476, 1992.

[35] S. Chaudhuri, S. Gulwani, R. Lublinerman, and S. Navidpour, "Proving programs robust," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pp. 102–112, ACM, 2011.

[36] H. Gargama and S. K. Chaturvedi, "Criticality assessment models for failure mode effects and criticality analysis using fuzzy logic," *IEEE Transactions on Reliability*, vol. 60, no. 1, pp. 102–110, 2011.

[37] S. V. Ukkusuri and W. F. Yushimito, "A methodology to assess the criticality of highway transportation networks," *Journal of Transportation Security*, vol. 2, no. 1-2, pp. 29–46, 2009.

[38] K. Lakshmanan, D. De Niz, R. Rajkumar, and G. Moreno, "Resource allocation in distributed mixed-criticality cyber-physical systems," in *Distributed Computing Systems (ICDCS), 2010 IEEE 30th International Conference on*, pp. 169–178, IEEE, 2010.

[39] K.-J. Park, D. M. Shrestha, Y.-B. Ko, N. H. Vaidya, and L. Sha, "IEEE 802.11 WLAN for medical-grade QoS," in *Proceedings of the 1st ACM international workshop on Medical-grade wireless networks*, pp. 3–8, ACM, 2009.

[40] J. C. Sanchez and J. P. Snover, "Automatic fault injection into a JAVA virtual machine (JVM)," Nov. 5 2002. US Patent 6,477,666.

[41] B. S. Baker, "Approximation algorithms for NP-complete problems on planar graphs," *Journal of the ACM (JACM)*, vol. 41, no. 1, pp. 153–180, 1994.

[42] J. Giesen, E. Schuberth, and M. Stojaković, "Approximate sorting," in *LATIN 2006: Theoretical Informatics*, pp. 524–531, Springer, 2006.

[43] S. Chaudhuri, S. Gulwani, and R. Lublinerman, "Continuity and robustness of programs," *Communications of the ACM*, vol. 55, no. 8, pp. 107–115, 2012.

[44] U. Ferraro-Petrillo, I. Finocchi, and G. F. Italiano, "The price of resiliency: a case study on sorting with memory faults," *Algorithmica*, vol. 53, no. 4, pp. 597–620, 2009.

[45] J. Xiang, L. Ye, E. Vicario, K. Tadano, and F. Machida, "Analysis of relevance and importance of components in system reliability," in *2015 2nd International Symposium on Dependable Computing and Internet of Things (DCIT)*, pp. 146–147, IEEE, 2015.

[46] D. L. Levine, C. D. Gill, and D. C. Schmidt, "Dynamic scheduling strategies for avionics mission computing," in *Digital Avionics Systems Conference, 1998. Proceedings., 17th DASC. The AIAA/IEEE/SAE*, vol. 1, pp. C15–1, IEEE, 1998.

[47] J. Cámara and R. de Lemos, "Evaluation of resilience in self-adaptive systems using probabilistic model-checking," in *Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pp. 53–62, IEEE Press, 2012.

[48] J. Dantas, R. Matos, J. Araujo, D. Oliveira, A. Oliveira, and P. Maciel, "Hierarchical model and sensitivity analysis for a cloud-based VoD streaming service," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, Workshop*, pp. 10–16, IEEE, 2016.

[49] G. J. Pai and J. B. Dugan, "Empirical analysis of software fault content and fault proneness using Bayesian methods," *IEEE Transactions on software Engineering*, vol. 33, no. 10, 2007.

[50] M. Piancó, B. Fonseca, and N. Antunes, "Code change history and software vulnerabilities," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, Workshop*, pp. 6–9, IEEE, 2016.

[51] I. Rodrigues, M. Ribeiro, F. Medeiros, P. Borba, B. Fonseca, and R. Gheyi, "Assessing fine-grained feature dependencies," *Information and Software Technology*, vol. 78, pp. 27–52, 2016.

[52] C. Lunardi, H. Quinn, L. Monroe, D. Oliveira, P. Navaux, and P. Rech, "Experimental and analytical analysis of sorting algorithms error criticality for HPC and large servers applications," *IEEE Transactions on Nuclear Science*, vol. 64, no. 8, pp. 2169–2178, 2017.

[53] S. Assaf and E. Upfal, "Fault tolerant sorting networks," *SIAM Journal on Discrete Mathematics*, vol. 4, no. 4, pp. 472–480, 1991.

[54] S. Guo, H.-Z. Huang, Z. Wang, and M. Xie, "Grid service reliability modeling and optimal task scheduling considering fault recovery," *IEEE Transactions on Reliability*, vol. 60, no. 1, pp. 263–274, 2011.

[55] M. De Kruijf, S. Nomura, and K. Sankaralingam, "Relax: An architectural framework for software recovery of hardware faults," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, pp. 497–508, 2010.

[56] S. Hukerikar and R. F. Lucas, "Rolex: Resilience-oriented language extensions for extreme-scale systems," *The Journal of Supercomputing*, vol. 72, no. 12, pp. 4662–4695, 2016.

[57] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "EnerJ: Approximate data types for safe and general low-power computation," in *ACM SIGPLAN Notices*, vol. 46, pp. 164–174, ACM, 2011.

[58] P. M. Wells, K. Chakraborty, and G. S. Sohi, "Adapting to intermittent faults in multicore systems," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 2, pp. 255–264, 2008.

[59] C. Constantinescu, "Trends and challenges in VLSI circuit reliability," *IEEE micro*, vol. 23, no. 4, pp. 14–19, 2003.

[60] C. Constantinescu, "Intermittent faults and effects on reliability of integrated circuits," in *Reliability and Maintainability Symposium, 2008. RAMS 2008. Annual*, pp. 370–374, IEEE, 2008.

[61] B. Schroeder and G. A. Gibson, "A large-scale study of failures in high-performance computing systems," *Dependable and Secure Computing, IEEE Transactions on*, vol. 7, no. 4, pp. 337–350, 2010.

[62] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, *et al.*, "Exascale computing study: Technology challenges in achieving exascale systems," 2008.

[63] A. B. Kahng, S. Kang, R. Kumar, and J. Sartori, "Slack redistribution for graceful degradation under voltage overscaling," in *Proceedings of the 2010 Asia and South Pacific Design Automation Conference*, pp. 825–831, IEEE Press, 2010.

[64] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, no. 4, pp. 75–82, 1997.

[65] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[66] A. Holler, G. Macher, T. Rauter, J. Iber, and C. Kreiner, "A virtual fault injection framework for reliability-aware software development," in *2015 IEEE International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pp. 69–74, IEEE, 2015.

[67] K. Parasyris, G. Tziantzoulis, C. D. Antonopoulos, and N. Bellas, "GemFI: A fault injection tool for studying the behavior of applications on unreliable substrates," in *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, pp. 622–629, IEEE, 2014.

[68] B. Atkinson, N. DeBardeleben, Q. Guan, R. Robey, and W. M. Jones, "Fault injection experiments with the CLAMR hydrodynamics mini-app," in *2014 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pp. 6–9, IEEE, 2014.

[69] R. Natella, D. Cotroneo, J. A. Duraes, and H. S. Madeira, "On fault representativeness of software fault injection," *IEEE Transactions on Software Engineering*, vol. 39, no. 1, pp. 80–96, 2013.

[70] J. Arlat, Y. Crouzet, J. Karlsson, P. Folkesson, E. Fuchs, and G. H. Leber, "Comparison of physical and software-implemented fault injection techniques," *IEEE Transactions on Computers*, vol. 52, no. 9, pp. 1115–1133, 2003.

[71] C. Areias, J. C. Cunha, and M. Vieira, "Studying the propagation of failures in SOAs," in *2015 IEEE International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pp. 81–86, IEEE, 2015.

[72] I. Irrera and M. Vieira, "Towards assessing representativeness of fault injection-generated failure data for online failure prediction," in *2015 IEEE International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pp. 75–80, IEEE, 2015.

[73] G. Canivet, P. Maistri, R. Leveugle, J. Clédière, F. Valette, and M. Renaudin, "Glitch and laser fault attacks onto a secure AES implementation on a SRAM-based FPGA," *Journal of cryptology*, vol. 24, no. 2, pp. 247–268, 2011.

[74] J. G. van Woudenberg, M. F. Witteman, and F. Menarini, "Practical optical fault injection on secure microcontrollers," in *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2011 Workshop on*, pp. 91–99, IEEE, 2011.

[75] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan, "The sorcerer's apprentice guide to fault attacks," *Proceedings of the IEEE*, vol. 94, no. 2, pp. 370–382, 2006.

[76] C. Meadows, "A cost-based framework for analysis of denial of service in networks," *Journal of Computer Security*, vol. 9, no. 1, pp. 143–164, 2001.

[77] C. Herley, "Security, cybercrime, and scale," *Communications of the ACM*, vol. 57, no. 9, pp. 64–71, 2014.

[78] D. H. Ackley, "Beyond efficiency," *Communications of the ACM*, vol. 56, no. 10, pp. 38–40, 2013.

[79] J. S. Monson, M. Wirthlin, and B. Hutchings, "A fault injection analysis of linux operating on an FPGA-embedded platform," *International Journal of Reconfigurable Computing*, vol. 2012, p. 7, 2012.

[80] R. Ghosh, K. S. Trivedi, V. K. Naik, and D. S. Kim, "End-to-end performability analysis for infrastructure-as-a-service cloud: An interacting stochastic models approach," in *Dependable Computing (PRDC), 2010 IEEE 16th Pacific Rim International Symposium on*, pp. 125–132, IEEE, 2010.

[81] J. P. Sterbenz, E. K. Çetinkaya, M. A. Hameed, A. Jabbar, S. Qian, and J. P. Rohrer, "Evaluation of network resilience, survivability, and disruption tolerance: analysis, topology generation, simulation, and experimentation," *Telecommunication systems*, vol. 52, no. 2, pp. 705–736, 2013.

[82] J. Elliott, M. Hoemmen, and F. Mueller, "Resilience in numerical methods: A position on fault models and methodologies," *arXiv preprint arXiv:1401.3013*, 2014.

[83] G. Gay, S. Rayadurgam, and M. P. Heimdahl, "Automated steering of model-based test oracles to admit real program behaviors," *IEEE Transactions on Software Engineering*, vol. 43, no. 6, pp. 531–555, 2017.

[84] SSBSE, *Search based software engineering: 4. international symposium, SSBSE 2012, Riva del Garda, Italy, September 28-30, 2012; proceedings.* Springer, 2012.

[85] K. V. Palem, L. N. Chakrapani, Z. M. Kedem, A. Lingamneni, and K. K. Muntimadugu, "Sustaining Moore's law in embedded computing through probabilistic and approximate design: retrospects and prospects," in *Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems*, pp. 1–10, ACM, 2009.

[86] P. Dagum and M. Luby, "Approximating probabilistic inference in Bayesian belief networks is NP-hard," *Artificial intelligence*, vol. 60, no. 1, pp. 141–153, 1993.

[87] J. Sloan, D. Kesler, R. Kumar, and A. Rahimi, "A numerical optimization-based methodology for application robustification: Transforming applications for error tolerance," in *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*, pp. 161–170, IEEE, 2010.

[88] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *ACM Sigplan Notices*, vol. 42, no. 6, pp. 89–100, 2007.

[89] L. Rudolph, "A robust sorting network," *IEEE Transactions on Computers*, vol. 100, no. 4, pp. 326–335, 1985.

[90] D. Fiala, F. Mueller, C. Engelmann, R. Riesen, K. Ferreira, and R. Brightwell, "Detection and correction of silent data corruption for large-scale high-performance computing," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, p. 78, IEEE Computer Society Press, 2012.

[91] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of AspectJ," in *European Conference on Object-Oriented Programming*, pp. 327–354, Springer, 2001.

[92] C. Borchert, H. Schirmeier, and O. Spinczyk, "Protecting the dynamic dispatch in C++ by dependability aspects.," in *GI-Jahrestagung*, pp. 521–536, 2012.

[93] C. A. Hoare, "Quicksort," *The Computer Journal*, vol. 5, no. 1, pp. 10–16, 1962.

[94] O. Astrachan, "Bubble sort: an archaeological algorithmic analysis," in *ACM SIGCSE Bulletin*, vol. 35, pp. 1–5, ACM, 2003.

[95] P. Diaconis and R. L. Graham, "Spearman's footrule as a measure of disarray," *Journal of the Royal Statistical Society. Series B (Methodological)*, pp. 262–268, 1977.

[96] A. Karatsuba and Y. Ofman, "Multiplication of multidigit numbers on automata," in *Soviet physics doklady*, vol. 7, p. 595, 1963.

[97] R. P. Brent, "Algorithms for matrix multiplication," tech. rep., STANFORD UNIV CA DEPT OF COMPUTER SCIENCE, 1970.

[98] S. Mathew and J. Varia, "Overview of Amazon Web Services," *Amazon Whitepapers*, 2014.

[99] R. L. Dobrushin and S. Ortyukov, "Lower bound for the redundancy of self-correcting arrangements of unreliable functional elements," *Problemy Peredachi Informatsii*, vol. 13, no. 1, pp. 82–89, 1977.

[100] P. Gács and A. Gál, "Lower bounds for the complexity of reliable boolean circuits with noisy gates," *IEEE Transactions on Information Theory*, vol. 40, no. 2, pp. 579–583, 1994.

[101] P. L. Chebyshev, "Des valeurs moyennes," *J. Math. Pures Appl.*, vol. 12, pp. 177–184, 1867.

[102] W. D. Orcutt, *Official Lawn Tennis Bulletin*, vol. 4. The Editors, 1897.

[103] M. T. Goodrich, "Spin-the-bottle sort and annealing sort: Oblivious sorting via round-robin random comparisons," *Algorithmica*, vol. 68, no. 4, pp. 835–858, 2014.

[104] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Introduction to algorithms second edition," 2001.

[105] A. L. Custódio, H. Rocha, and L. N. Vicente, "Incorporating minimum Frobenius norm models in direct search," *Computational Optimization and Applications*, vol. 46, no. 2, pp. 265–278, 2010.

[106] G. Ballard, A. R. Benson, A. Druinsky, B. Lipshitz, and O. Schwartz, "Improving the numerical stability of fast matrix multiplication," *arXiv preprint arXiv:1507.00687*, 2015.

[107] B. Baudry, F. Fleurey, J.-M. Jzquel, and Y. L. Traon, "From genetic to bacteriological algorithms for mutation-based testing: Research articles," *Verification and Reliability Software Testing*, vol. 15, pp. 73–96, June 2005.

[108] J. Campos, Y. Ge, G. Fraser, M. Eler, and A. Arcuri, "An empirical evaluation of evolutionary algorithms for test suite generation," in *International Symposium on Search Based Software Engineering*, pp. 33–48, Springer, 2017.

[109] J. Kukunas, R. D. Cupper, and G. M. Kapfhammer, "A genetic algorithm to improve linux kernel performance on resource-constrained devices," in *Proceedings of the 12th annual conference companion on Genetic and evolutionary computation*, pp. 2095–2096, ACM, 2010.

[110] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper, "Testability transformation," *IEEE Transactions on Software Engineering*, vol. 30, no. 1, pp. 3–16, 2004.

[111] A. Arcuri, M. Z. Iqbal, and L. Briand, "Black-box system testing of real-time embedded systems using random and search-based testing," in *IFIP International Conference on Testing Software and Systems*, pp. 95–110, Springer, 2010.

[112] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, p. 436, 2015.

[113] M. Ancona, E. Ceolini, C. Öztireli, and M. Gross, "A unified view of gradient-based attribution methods for deep neural networks," *arXiv preprint arXiv:1711.06104*, 2017.