DEVELOPER-CENTRIC AUTOMATED DEBUGGING

A Dissertation
Presented to
The Academic Faculty

By

Xiangyu Li

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
College of Computing
School of Computer Science

Georgia Institute of Technology

May 2021

© Xiangyu Li 2021

DEVELOPER-CENTRIC AUTOMATED DEBUGGING

Thesis committee:

Dr. Alessandro Orso
School of Computer Science
Georgia Institute of Technology

Dr. Spencer Rugaber
School of Computer Science
Georgia Institute of Technology

Dr. David Devecsery
School of Computer Science
Georgia Institute of Technology

Dr. Marcelo d'Amorim
Department of Computer Science
Federal University of Pernambuco

Dr. Qirun Zhang
School of Computer Science
Georgia Institute of Technology

Date approved: 26 April 2021

ACKNOWLEDGMENTS

First, I would like to thank my advisor, Professor Alessandro Orso, for his precious advices on my research and tremendous support. His guidance helped me develop my research projects that lead to this thesis, and I learned a lot from him through this journey. I also would like to thank my committee members for their insightful comments that helped me improve my thesis.

I express my utmost gratitude to my former advisor, Professor Mary Jean Harrold, for her unlimited support at the beginning of my Ph.D. study. Although she is no longer with us, her kindness, enthusiasm, and academic excellence continues to inspire me.

I would like to thank the current and past members of the Arktos research group. They gave me many great inspirations for research ideas and made my PhD study considerably more enjoyable.

My special thanks go to my family for their unconditional support and love.

TABLE OF CONTENTS

Acknowledgments iii

List of Tables vii

List of Figures viii

Summary x

Chapter 1: Introduction and Motivation 1

1.1 Thesis Statement 2

1.2 Approaches 3

1.3 Contributions 8

1.4 Organization 9

Chapter 2: Background 11

2.1 Statistical Fault Localization 11

2.2 Dynamic Program Slicing 12

Chapter 3: Overall Vision 15

Chapter 4: Related Work 18

4.1 Interactive or Automated Debugging Techniques 18

4.2 Program Slicing and Dynamic Dependency Analysis 21

Chapter 5: Enlightened Debugging 24

5.1 Technique 24

5.1.1 Test Runner & Dependency Analyzer 26

5.1.2 SFL Calculator 26

5.1.3 Query Generator 27

5.1.4 Feedback Analyzer 29

5.2 Illustrative Example 31

5.3 Empirical Evaluation 34

5.3.1 Study with Simulated Users 35

5.3.2 User Study 42

5.3.3 Limitations and Threats to Validity 47

Chapter 6: More Accurate Dynamic Slicing for Better Supporting Software Debugging 49

6.1 Technique 49

6.1.1 Terminology and Definitions 49

6.1.2 Motivating Examples 50

6.1.3 Computing PMDs 52

6.2 Empirical Evaluation 61

6.2.1 Experiment Setup 61

6.2.2 RQ1: Do PMD slices include (real-world) faults that traditional dynamic slices would miss? 63

6.2.3 RQ2: How much larger are PMD slices compared to traditional dynamic slices? . . . 66

6.2.4 RQ3: What is the computational cost of PMD slicer compared to a traditional dynamic slicer? . . . 68

6.2.5 Limitations and Threats to Validity . . . 70

Chapter 7: TESSERACT: Scalable Dependency-Based Interactive Debugging . . 71

7.1 Technique . . . 71

7.1.1 Illustrative Example . . . 74

7.1.2 Recorder and Partitioner . . . 75

7.1.3 Comprehensive Analyzer . . . 76

7.1.4 Inter-epoch Analyzer . . . 81

7.1.5 Implementation . . . 87

7.2 Empirical Evaluation . . . 87

7.2.1 Experiment Setup . . . 88

7.2.2 RQ1: How well does TESSERACT scale on the benchmark executions? . . . 89

7.2.3 RQ2: How well does the Comprehensive Analyzer support interactive debugging? . . . 93

7.2.4 Limitations and Threats to Validity . . . 95

Chapter 8: Conclusion . . . 97

References . . . 100

LIST OF TABLES

5.1 Benchmarks and faults considered for ENLIGHTEN. 36

5.2 Summary of results for real faults for ENLIGHTEN. 38

5.3 Summary of results for mutation faults for ENLIGHTEN. 39

5.4 Sensitivity of ENLIGHTEN to human errors. Values indicate the percentual increase in the number of queries over an ideal oracle (i.e., a user that does not make mistakes). 42

5.5 Debugging tasks for the user study for ENLIGHTEN. 43

5.6 Results for User Study 1 for ENLIGHTEN. 45

5.7 Results for User Study 2 for ENLIGHTEN. 47

6.1 Benchmark programs for PMD. 62

6.2 Summary of cases that the faults are not in dynamic slices but are in PMD slices. 64

6.3 Size comparison for PMD slices. 66

6.4 Running time comparison for PMD slicer. 67

7.1 Benchmark executions for TESSERACT. 88

7.2 Running time of Inter-epoch Analyzer. 91

7.3 Storage overhead of Inter-epoch Analyzer. 91

7.4 Response time and memory consumption of Comprehensive Analyzer. . . . 94

LIST OF FIGURES

2.1 Relevant slicing example. 13

3.1 Debugging workflow. 16

5.1 Overview of ENLIGHTEN. 25

5.2 Algorithm for incorporating feedback in ENLIGHTEN. 30

5.3 Stack and corresponding test suite example for ENLIGHTEN. 32

5.4 Debugging query on the 1st iteration. 34

5.5 Coverage matrices before / after the 1st iteration. 34

6.1 Example of PMD caused by array writes. 51

6.2 Example of PMD caused by array reads. 52

6.3 Algorithm for computing PMDs due to array writes. 54

6.4 Computing PMDs for the first motivating example. 55

6.5 Algorithm for computing PMDs due to array reads. 60

7.1 Overview of TESSERACT. 72

7.2 Example execution for TESSERACT. 74

7.3 Control dependency stack of example for TESSERACT. 79

7.4 Historical states of example execution for TESSERACT. 80

7.5 Example of tree of epoch ranges for parallel scan. 84

7.6 Merging local analysis results of storage locations. 85

7.7 Merging inter-epoch frames. 85

7.8 Speedup ratio of Inter-epoch Analyzer from 1 to 256 cores. 90

SUMMARY

Software debugging is an expensive activity that is responsible for a significant part of software maintenance cost. In particular, locating faulty code (i.e., fault localization) is one of the most challenging parts of software debugging. In the past years, researchers have proposed many techniques that aim at fully automating the task of fault localization. Although these techniques have been shown to be effective in reducing the amount of code developers need to inspect to locate faults, there is growing evidence that they provide developers with limited help in realistic debugging scenarios. I believe that a practical automated debugging technique should have human developers at the center of the debugging process rather than trying to completely replace them.

In this dissertation, I present three of my techniques that support developer-centric automated debugging. First, I present ENLIGHTEN, an interactive, feedback-driven fault localization technique. ENLIGHTEN supports and automates developers' debugging workflow as follows. It 1) uses traditional statistical fault localization (SFL) to formulate an initial hypothesis of where the fault may be; 2) identifies a relevant subset of execution that can help support or refute the formulated hypothesis; 3) presents the developer with a query about the identified execution subset in the form of a correctness question about the input-output relation of the partial execution; 4) refines its hypothesis of the fault by using the developer's feedback; and 5) repeats these steps until the fault is found. Second, I discuss my work on improving the accuracy of dynamic dependence analysis, which is a powerful tool for developers to investigate program behavior in an interactive debugging setting and a foundation that many automated debugging techniques leverage to model dynamic execution semantics. I present my finding that existing dynamic dependence analysis techniques could miss the cause-effect relations between faults and the observed failures if the faulty program states propagate via incorrect computation of memory addresses. To address this limitation, I define the concept of potential memory-address dependence, which explicitly

represents this type of causal relations, and describe an algorithm that computes it. Third, I present TESSERACT, a technique that improves the scalability of dynamic dependency analysis in the context of interactive debugging. Many existing dependency-based debugging techniques are shown to work well on short executions, but fail to scale to larger ones. TESSERACT has the potential to address this limitation by utilizing a record-and-replay system to efficiently recreate the failing execution, break it down into small time slices, and analyze these slices in a parallelized, on-demand fashion.

CHAPTER 1
INTRODUCTION AND MOTIVATION

Software debugging is a notoriously difficult, and time-consuming activity. In particular, the task of locating faulty code (i.e., fault localization) is one the most challenging parts.

For this reason, researchers have proposed numerous techniques to help developers reduce the cost of fault localization and debugging in general (e.g., [1, 2, 3, 4, 5, 6, 7, 8]). Past research of fully automated fault localization techniques [3, 9, 10, 11] shows that they can effectively reduce the scopes that developers need to explore to find the faults. However, there is evidence that they offer developers limited help in realistic debugging tasks [12, 13]. On the one hand, fully automated fault localization techniques usually fail to provide comprehensive information for developers to understand the faults. Many of these techniques output only a list of potentially faulty program entities (e.g., statements, blocks, and predicates). Previous studies [12, 13] of their empirical benefits has shown that it is unrealistic to assume that developers provided with a possibly long list of suspicious statements would go through this list in order and immediately spot the fault when they see it without additional context. On the other hand, fully automated fault localization techniques often do not have sufficient information about the failures and the expected behavior of the programs to make precise inference of what and where the faults are [14]. In real-world software development environments, significant parts of software specifications exist only in the developers' minds. Although some fully automated techniques [15] have been adopted in practice, they are limited to a narrow range of debugging scenarios, such as isolating crash-inducing inputs and locating domain-specific faults.

Today, developers usually still have to manually diagnose software failures with symbolic debuggers, logs, and print statements [16], which only provide ways of inspecting system states but leave analyzing and reasoning about program behavior to the developers.

However, even extremely short program executions, such as unit tests, often contain a large number of executed statements that is beyond human developers' capability to fully analyze. Performing debugging activities efficiently therefore usually requires developers to make good hypotheses about the faults and select the right parts of the failing executions to inspect. For these reasons, debugging remains an inefficient trial-and-error process. There is a clear gap between automated debugging research and real-world developer needs.

My dissertation work focuses on bridging this gap by combining the power of automated analyses and human developers' rich knowledge of software specifications in a debugging workflow that highly automates debugging activities and, at the same time, involves developers throughout the process. This combination creates synergistic benefits for debugging as it utilizes the strengths of both sides while mitigating their weaknesses. On the one hand, automated analyses is able to collect and analyze the vast amount of information contained in the executions of the system under test (SUT), identify the suspicious program entities that might be responsible for the failure, and communicate the analysis results to the developers through suitably designed abstractions. This enhances developers' ability to inspect and understand program behavior and relieves them from making unreliable hypothesis about the fault. On the other hand, involving developers throughout the debugging process allows them to give information about software specifications to the automated tools, as that information is needed in an interactive fashion, which could improve the accuracy of the analyses.

1.1 Thesis Statement

To address some of the limitations of existing debugging techniques and bridge the gap between automated debugging research and the current software development practice, my dissertation work develops techniques that supports a developer-centric automated debugging approach. Different from traditional automated debugging techniques that try to completely automate the task and replace the developers, my approach combines automated

analyses and human intelligence in a more engaging debugging process. Through the use of my techniques, we can build better developer tools that help developers locate faults more efficiently and understand dynamic program behavior better for real-world programs and failing executions.

1.2 Approaches

In Chapter 5, I present ENLIGHTEN, an interactive, feedback-driven fault localization technique. I defined ENLIGHTEN so as to follow the way in which debugging is typically performed, with the goal of helping—rather than completely replacing—the humans in the loop. Typically, developers would study the failure at hand, make hypotheses on what the cause(s) of the failure may be, and examine a subset of the execution that can confirm or disprove their hypothesis. They would then leverage the additional understanding of the failure acquired in this process to make new hypotheses or refine the existing ones, examine additional subsets of the execution, and so on. This process would continue until either the developers give up or they find the fault.

ENLIGHTEN aims to mimic and support this process as follows. First, it uses traditional statistical fault localization (SFL) to formulate an initial hypothesis of where the fault may be. Second, it identifies a relevant subset of the execution that can help support or negate the formulated hypothesis. Intuitively, this execution subset is identified in the form of a method invocation that results in the execution of highly suspicious entities. Third, ENLIGHTEN presents the developer with a query about the identified method invocation, expressed in terms of the input and resulting output of the invocation. Fourth, ENLIGHTEN collects the developer feedback on the correctness of individual data values in the provided inputs and outputs and encodes this feedback as extra program specifications. Finally, ENLIGHTEN repeats these steps until the fault is found or the developer decides to stop.

This approach can overcome the important limitations of traditional SFL. Specifically,

ENLIGHTEN does not require developers to decide whether a statement in isolation is correct, but rather to check high-level input-output relationships at the method level. We believe that operating at the level of abstraction of a method, whose semantics is often well understood by developers, can make the technique considerably more effective and usable. Moreover, developers can skip queries that they cannot answer and, as shown in our evaluation, the technique is resilient to occasional erroneous responses. Basically, when successful, ENLIGHTEN can nicely guide the developers towards the fault by following an iterative process that gets their input at a level of abstraction they can typically understand.

To evaluate ENLIGHTEN, I implemented the approach and performed two empirical studies. In the first study, I used an automated oracle to simulate the presence of a developer and applied ENLIGHTEN to a large number of faults in 3 open source programs. The faults considered included 27 real faults and 1,780 mutation faults, which I generated to increase the number of data points. As the results show, for over 96% of the faults considered, ENLIGHTEN required less than 10 interactions with the simulated user to localize the fault. In the second part of the evaluation, I performed an actual user study. I selected 4 real faults and 24 participants and assigned to each participant two debugging tasks: one to be performed using ENLIGHTEN, and one to be performed using the debugging technique(s) of their choice. When using ENLIGHTEN, the participants performed considerably better than when debugging without the help of my tool. The improvement was significant in terms of both number of faults localized and time needed to localize the faults. Overall, the results provide a clear indication that ENLIGHTEN is a promising approach for debugging and fault localization.

On the conceptual level, the study of ENLIGHTEN indicates that, despite the usually huge size of the DDG, combining automated analysis with developers' knowledge of the software specification enables quick convergence to the faults during the exploration of the DDG. From the participants' comments for the user study of ENLIGHTEN, we also learned that showing developers the concrete program states is crucial for developers to understand

the program behaviors and provides useful feedback to the automated debugging tool.

While Enlighten automates a significant part of the debugging process, it leaves to the developer the task of understanding the behavior of the selected method invocations and specifying the correctness of the suspicious variable values. One promising research direction that helps developers perform this task is the dependency-based approach [17, 18], which enables developers to explore the execution by following dynamic dependencies. One major limitation of using dynamic dependences for this purpose is that it might miss some cause-effect relations between the faults and the observed faulty data items. Missing such relations reduces the effectiveness of the dependency-based debugging approach, and in the worst case, leads the debugging process to dead ends. To address this limitation, researchers proposed relevant slicing techniques [19, 20], which aims to discover not only the actual dynamic dependences of the slicing criterion, but also its potential dependences, which could have affected the value of the slicing criterion if they are evaluated differently. In Chapter 6, I present my finding that relevant slicing still misses a class of potential dependences that are caused by memory address computation and describe my approach to compute these potential dependences.

Specifically, traditional dynamic dependence analysis would miss the potential dependence between two instruction instances s_1 and s_2 such that (1) s_1 directly or indirectly affects the computation of a memory-address ma, (2) ma is used to identify a memory location in an assignment, (3) s_2 reads the value of a memory location m different from that identified by ma, and (4) ma could point to m had the program execution been different. As an example, consider (1) an instruction s_1 that defines a value $index$, (2) a later instruction s_{def} that uses $index$ as the index of an array element to modify (`a[index] = newValue`), (3) an instruction s_2 that reads the value of the same array but at a different index (`a[index']`). Although there are no actual dependences between s_1 and s_2, s_1 potentially affects the behavior of s_2; had s_1 assigned to $index$ the value of $index'$, s_2 would have accessed the element of the array that was defined in s_{def}. We define this kind of

dependence between s_1 and s_2 as a *potential memory-address dependence* (PMD).

Considering PMDs is important in debugging because they allow developers to detect cases in which the result of the assignment in s_1 is incorrect (because either s_1 is faulty or some operands used in s_1 are incorrect). Although relevant slicing can detect missing assignments due to incorrect control flow, it does not account for PMDs. Intuitively, whereas relevant slicing identifies missing dependences between a faulty data item [1] m and predicates that are not affecting m but should have in a correct execution, PMDs represent missing dependences between m and assignments that are not affecting m but should have in a correct execution.

To evaluate the work, I implemented the PMD slicer which considers PMDs. I then used the PMD slicer to assess the effectiveness of my approach in the context of debugging by applying it to a benchmark of 364 real faults and 880 failing tests that reveal them. In the evaluation, I compared my technique with a traditional dynamic slicing technique by (1) computing slices for each failing execution from the point of failure using both techniques and (2) checking whether the slices generated by the two techniques included the fault. The results are encouraging and show that my technique is useful: 9.2% (81) of the failing tests produced faulty values for which traditional dynamic slices did not contain the corresponding faults, while the slices generated by PMD slicer did. I also compared the sizes of the slices produced by the PMD slicer with the sizes of the corresponding slices produced by the traditional dynamic slicer. As the results show, my technique only moderately increases slice sizes: on average, the increase is 7.2% in the number of static instructions and 10.1% in the number of dynamic instruction instances.

My work on enlightened debugging and the potential memory-address dependence both relies on the dynamic dependency analysis, which introduces a high computational cost [21, 22] that is a major challenge of applying these techniques in realistic debugging scenarios. On one hand, the dynamic analysis usually slows down the execution by around two orders

[1] I use the term ”data item” to denote the state of a memory or register location at a certain time point.

of magnitude, leading to long waits for the users. On the other hand, representing the dynamic dependencies consumes extremely large amount of memory. On our benchmark of x86 binaries, one second of program execution generates a dynamic dependency graph (DDG) of several billion nodes, which exceeds what the memory of a typical development machine can hold. For these reasons, current dependency-based debugging techniques work on only short executions with DDGs of at most a few gigabytes.

To address this limitation of existing approaches, I propose TESSERACT, a technique that supports a dependency-based debugging process for realistic program executions by utilizing parallelism and the power of computing clusters. TESSERACT relies on a record-replay approach to store the information needed to reproduce the execution with low overhead in terms of both execution time and storage space. Specifically, it decomposes the recorded execution into small time slices called epochs that can be analyzed one at a time, which allows TESSERACT to handle otherwise non-scalable dynamic analyses. TESSERACT then runs the analyses on the epochs in parallel on a computing cluster, which enables the technique not only to compute the information needed faster, but also to handle larger amount of data. To reduce the overall cost of analyzing the execution, TESSERACT performs two dynamic analyses: the inter-epoch and the comprehensive analyses. The *inter-epoch analysis* runs on all epochs in parallel and computes *inter-epoch dependencies* that are used to identify the specific epochs that are actually needed during the interactive debugging session. The *comprehensive analysis*, which produces all the information necessary to support the debugging workflow inside an epoch, is applied on-demand on those needed epochs only.

TESSERACT enables developers to trace dynamic dependencies in a way similar to the approach in Whyline [21]. For each data item in the program state, the technique provides the developer with an option to go to its dynamic reaching definition, which is the statement instance that computed the value of the data item; and for each executed statement instance, the technique provides an option to go to its dynamic control dependency predecessor.

After the developer selects a navigation option, TESSERACT not only shows the target statement instance, but also restores the full program state to the moment right before the statement instance was executed, as I believe that concrete values of the program state are essential to help developers understand program behavior.

To evaluate TESSERACT, I developed a prototype implementation that support debugging x86 binaries and performed an empirical study on executions of real-world programs. To investigate how well TESSERACT works in different scenarios, I considered executions of varying lengths ranging from 8 to 2460 milliseconds. I measured the running time of the inter-epoch analysis, which is a factor of developers' wait time when exploring parts of the execution that cross epoch boundaries, on different number of CPU cores. The results are promising, in that they show that TESSERACT scales well with the number of available cores. For the benchmark executions longer than 400 milliseconds and on a cluster of 256 cores, in particular, TESSERACT achieved speedups for the inter-epoch analysis between 18.5x to 100x, with an average of 59x, with respect to its execution on a single core. The speedups achieved for longer executions are even more significant and continue to scale beyond 256 cores. I also evaluated the response time of TESSERACT during an interactive debugging session. Our results are again encouraging, as TESSERACT was able to answer the vast majority of user queries with no noticeable delay.

Conceptually, TESSERACT investigates the feasibility of separately computing the inter-epoch dynamic dependencies of individual epochs and aggregating the results. The experiment results show that this approach is effective. On main reason that contributes to the effectiveness is that, compared to the complete DDG, the inter-epoch dynamic dependencies account for only a small part even for short epoch lengths of about 1 millisecond.

1.3 Contributions

This dissertation provides the following novel research contributions:

- ENLIGHTEN, an interactive, feedback-driven fault localization technique that sup-

ports developers' debugging workflow, and highly automates the debugging process.

- A prototype implementation of ENLIGHTEN that is publicly available [23].
- An empirical evaluation of the effective of ENLIGHTEN.
- The new concept of potential memory-address dependence (PMD), which allows for improving the applicability of dynamic dependence analysis and the techniques that relies on it in the presence of incorrect memory-address computations.
- An algorithm for computing precise potential memory-address dependences and a dynamic slicing technique that leverages this algorithm.
- An evaluation of the effectiveness of potential memory-address dependences on a large benchmark of real tests and faults, in terms of both effectiveness (i.e., ability to generate slices that include the fault) and efficiency (i.e., size of the generated slices).
- An implementation of the dynamic slicer that considers PMD, which is publicly available, together with the experiment data and infrastructure, at https://sites.google.com/view/pmd-artifacts/home
- TESSERACT, a new dependency-based debugging technique that can support real-world executions by utilizing massive parallelism and computing clusters.
- A tool that implements TESSERACT, supports debugging of x86 binaries, and is publicly available with our experiment data at https://sites.google.com/view/projecttesseract.
- An empirical evaluation of TESSERACT on a benchmark of executions of real-world programs.

1.4 Organization

The rest of the thesis dissertation is organized as follows. Chapter 2 introduces the background information needed in the dissertation. Chapter 3 discusses the overall vision of the

developer-centric automated debugging process and how each of my techniques supports it. Chapter 4 discusses related work. Chapter 5, Chapter 6 and Chapter 7 provides details on ENLIGHTEN, potential memory-address dependences, and TESSERACT, respectively. Chapter 8 provides a conclusion to the research.

CHAPTER 2
BACKGROUND

This chapter provides the necessary background information for the dissertation. It is organized as follows. Section 2.1 discusses statistical fault localization. Section 2.2 presents dynamic program slicing.

2.1 Statistical Fault Localization

Because of the high cost of software debugging, researchers have proposed numerous automated debugging techniques. Among them, one promising approach is statistical fault localization (SFL) [3, 11]. On the high level, an SFL technique takes as input the program under debugging and a set of tests, which contains both passing and failing cases, and outputs, for each program entity (i.e., statement, basic block, function, etc.), a suspiciousness score, which indicates the likelihood that the program entity is faulty. To do that, the technique executes the test cases on the program, observes the test outcomes and the coverage of each program entity in the tests, and performs a statistical inference to compute the suspiciousness for each entity by correlating its coverage and the occurrences of test failures. Intuitively, the inference considers program entities that are more frequently executed in failing test cases and less frequently executed in passing ones to be more suspicious. SFL techniques typically present the results to the developer as a list of program entities ranked in decreasing order of suspiciousness. The developer is then expected to inspect each program entity in that order to find the fault. Empirical evidence shows that SFL technique are usually effective in ranking the actual faults near the top of the ranked list.

Researchers have proposed numerous formulas for calculating the suspiciousness scores. I briefly describe the Ochiai formula [24] here as past empirical studies showed that it is among the most effective metrics [25]. For a program entity e, I denote the number of

passing test cases that covered e as a_{ep}, the number of failing test cases that covered e as a_{ef}, and the number of failing test cases that did not cover e as a_{nf}. The suspiciousness score of e is then calculated by Equation (2.1).

$$suspiciousness(e) = \frac{a_{ef}}{\sqrt{(a_{ef} + a_{ep}) \times (a_{ef} + a_{nf})}} \tag{2.1}$$

2.2 Dynamic Program Slicing

Debugging usually starts with observing an incorrect program output. To reduce the cost of debugging in this scenario, researchers have proposed *dynamic program slicing* [26, 27]. Given a faulty output value v at an execution position p, a dynamic slicing technique aims to reduce the amount of code that the developers need to inspect to find the fault by computing a subset of program statements that, either directly or indirectly, influenced v through dynamic data and control dependencies. Specifically, a statement instance s_1 is considered data dependent on another statement instance s_0 if s_1 uses as operands any variable values defined by s_0; and a statement instance s_3 is considered control dependent on a branching statement instance s_2 if the execution of s_3 is dependent on s_2 taking the particular branch in the execution (i.e., s_2 might not be executed if s_2 took a different branch).

Agrawal and Horgan formulated dynamic slicing as a graph reachability problem [27]. The technique first runs a *dynamic dependency analysis* on the execution history to produce a *dynamic dependency graph (DDG)*, and then computes dynamic slices on the graph. More formally, the dynamic dependency analysis outputs a graph $G = (V, E)$ with V being a set of vertices and $E \subseteq V \times V$ being a set of edges. The vertices represent the executed statement instances in the execution history and the edges represent the dynamic dependencies between them. Conceptually, the approach builds the DDG by processing each statement instance in the execution order. It computes data dependencies as follows. The technique keeps track of, for each storage location m (incl. memory locations and reg-

```
1 int var = 0;
2 boolean condition = false; // faulty. Should be condition = true
3 if (condition) {
4     var = 42;
5 }
6 print(var);
```

Figure 2.1: Relevant slicing example.

isters), its dynamic reaching definition $drd(m)$, which is the statement instance that made the most recent assignment to m. For each statement instance s, it computes a use set that contains the storage locations used by s as operands and a def set that contains the storage locations defined by s. The vertex for s is data dependent on the dynamic reaching definitions of all elements in its use set. The technique then updates the dynamic reaching definitions of the storage locations in the def set to s. The approach computes control dependencies inside each function call by keeping track of the most recently executed instances b of each branching statement B. When a statement s that is directly (static) control dependent on B is executed, the technique sets the control dependency of s as the saved instance b.

One main drawback of using dynamic slicing in debugging is that dynamic slices do not always contain the faults. It misses a class of faults when the faulty program states do not propagate through the dynamic control and data dependencies actually exercised in the failing execution. Instead, the faulty definitions in these cases are caused by statement instances that are not their dynamic dependency predecessors but potentially could have affected them if the program executed differently. To mitigate this limitation, researchers have proposed *relevant slicing* to compute potential dependencies that are caused by control flow choices. To illustrate, Figure 2.1 shows an example in which developers have to use relevant slices to find the fault. The code snippet has a fault on line 2. The variable `condition` is supposed to be `true`. Because of the fault, the program took the false branch one line 3 and skipped the intended assignment to the variable `var`. As a result, the print statement on line 6 outputs an incorrect value. Using the dynamic slice of the variable `var` on line 6 would miss the actual faulty statement as the value of `var` is only defined on

line 1 and has no dynamic dependencies on line 2. To address this issue, the relevant slicing technique observes that had the variable `condition` been assigned the value of `true`, the variable `var` would have a different value as the program would have taken the true branch on line 3. Therefore, the technique adds the executed instance of the branching statement on line 3 and its dynamic dependency predecessors, which include the actual fault on line 2, to the slice.

CHAPTER 3

OVERALL VISION

This chapter presents the overall vision of the developer-centric automated debugging process and how my techniques improves over existing debugging techniques to support it.

To better explain, I discuss my techniques in the context of a debugging workflow that is often adopted in the modern software development practice shown in Figure 3.1. The process usually starts with one or more failing executions, together with a possibly empty set of passing tests. To start, the developer would make an initial hypothesis of the fault. Based on this hypothesis, he/she selects a set of suspicious program entities that might be responsible for the failure. This includes, for example, suspicious statements, functions calls, and variables. The developer then inspects the execution, checking the selected entities against his/her understanding of the program specification. The result of the inspection is that either the fault is identified or the hypothesis is refuted. In the latter case, the developer gains a better understanding of the fault, revises his/her hypothesis of the fault, and repeats this process.

The first part of my work, Enlightened Debugging, aims at supporting this debugging workflow while automating a significant part of it. Specifically, it takes as input the program and the test cases, automatically generates a hypothesis of the fault, and presents to the developer the most suspicious method call and the likely incorrect variable values in its output program state. The developer then manually inspects the selected call and provides feedback to the technique by specifying the correctness of individual variables. Similar to a developer learning from inspecting the execution, the technique learns from the developer's feedback, revises its hypothesis of the fault, and repeats this process until the developer identifies the fault.

The Enlightened Debugging approach leaves to the developer the task of understanding

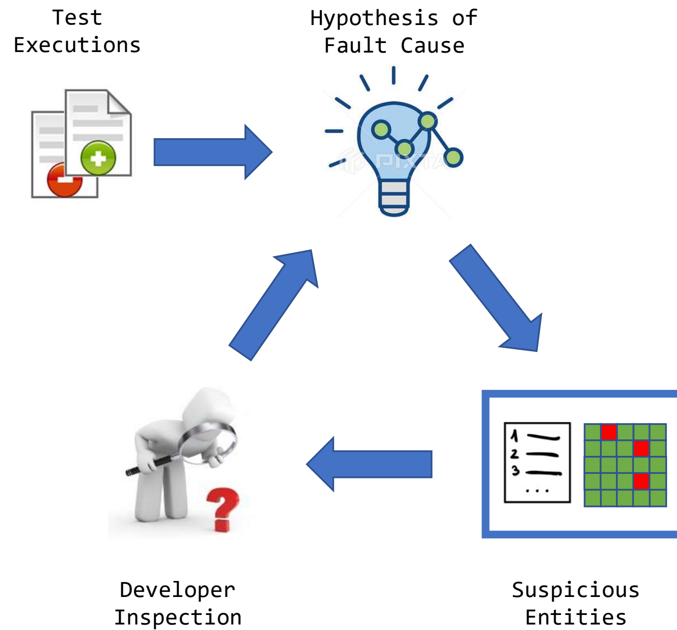

Figure 3.1: Debugging workflow.

the input-output relations in the selected method calls and specifying the correctness of the output variables for the given input. This approach has the advantage of allowing the technique to learn more about the program specification, which can not be fully encoded in test cases and is, in fact, mostly in the developer's mind. However, understanding the selected part of the execution is still non-trivial. I believe that a practical debugging technique should provide tools to support this task. One promising approach is to allow the developer to investigate the execution by exploring dynamic dependencies in the backward direction [21]. In the second part of my work, I introduced the potential memory-address dependencies (PMD) to address one major limitation of the existing dynamic dependency analysis techniques. Specifically, existing techniques could miss a class of potential dependencies that are caused by incorrect computation of memory addresses. My technique augments traditional DDGs with PMD relations. This provides developers with enhanced capability for understanding program behaviors. Moreover, it also gives automated fault localization techniques a more comprehensive picture of program dependencies.

To apply my techniques and other dependency-based debugging approaches in the real

world, one main bottleneck is the prohibitively high computational cost of dynamic dependency analysis. This leads to long waiting time before the developer can start inspecting faulty executions and long response time during the interactive debugging session. The third part of my work proposes Tesseract, which addresses this limitation by massively parallelizing dynamic dependency analysis on computing clusters. To do that, Tesseract splits the execution under debugging into small time slices (epochs), runs a light-weight dynamic analysis in parallel on all the epochs, and applies the expensive dynamic dependency analysis on demand on only the required epochs. This approach allows developers to immediately start debugging after observing a failure and, for our benchmark executions, reduces the response time of dependency-tracing requests to near zero in most cases.

I discuss the details of my techniques in Chapter 5, Chapter 6, and Chapter 7.

CHAPTER 4
RELATED WORK

The chapter describes the techniques that are the most closely related to my work in this thesis proposal, including interactive or automated debugging techniques, and dynamic and relevant slicing techniques.

4.1 Interactive or Automated Debugging Techniques

Algorithmic Debugging (AD) [1, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81] is an interactive debugging technique that was proposed in the functional programming community by Shapiro in the early eighties [2]. Similar to , AD asks developers questions on the correctness of specific function invocations in the execution tree for a given failing test. The tree is then systematically pruned based on the answers to these questions until the fault can be isolated. differs from AD in two important ways. First, AD uses basic heuristics to identify which function invocations to target, whereas leverages SFL and dynamic dependences. Second, AD requires developers to determine whether a function invocation is completely correct, which is difficult to do in the common case of functions that involve large portions of the program state. (This problem is common to most techniques based on AD [82, 83], including my own previous work [84], and tends to make these approaches error-prone and impractical.) Conversely, asks developers for feedback on individual input and output values, which I believe (and the results show) is a more realistic approach.

Francel and colleagues proposed an interactive debugging approach that combines algorithmic debugging and program slicing [85]. Similar to , in each iteration, the technique identifies a statement instance for developer inspection; the developer gives feedback to the tool about the correctness of the variable defined by the statement instance; and the

technique uses the feedback to revise its knowledge of the fault, resulting in a smaller set of potentially faulty statement instances. The technique reduces the candidate set of faulty statement instances by using dynamic dependency information. On one hand, all the statement instances that are the dynamic dependency predecessors of the correct variable values are considered to be correct and excluded from the candidate set. On the other hand, when the developer aims at finding only one single fault, the candidate set of faulty statement instances is restricted by the union of the dynamic dependency predecessor sets of the incorrect variable values. To reduce the fault candidate set efficiently, in each iteration the technique picks a statement instance that minimizes the expectation of resulting candidate set size assuming that the probabilities of the statement instance computing a correct and incorrect values are equal. is different from this debugging approach in several aspects. First, Francel's approach has the limitation that it might incorrectly exclude the actual fault from the candidate set, which would result in failing to locate the fault, if the faulty statement is a dynamic dependency predecessor of a correct variable value. eliminates this limitation by using a statistical analysis to compute the suspiciousness of statements. Second, compared to Francel's approach, which requires developers to determine the correctness of statement instances at arbitrary execution locations in the context of the whole program execution, asks about the correctness of variable values in the outputs of method calls in the context of those calls, which provides developers with richer information to answer the queries.

Statistical fault localization (SFL) [3, 86, 87, 9, 15, 10, 88, 89, 90, 91, 92, 11, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102] computes suspiciousness scores for code entities based on the strength of correlation between the execution of such code entities and the test results. Intuitively, code entities that are more frequently executed in failing executions are considered more likely to be faulty. Although SFL has been a disruptive change in the area of debugging, and has generated a tremendous amount of followup research, it has some significant limitations. Most SFL techniques tend to make strong, often unrealistic assumptions on how developers behave when debugging. In particular, previous work has

shown that it is unrealistic to assume that developers provided with a possibly long list of suspicious statements would go through this list in order and immediately spot the fault when they see it, without any additional context.

Gong and colleagues [4] proposed an interactive fault localization technique that continuously updates the ranked list of suspicious statements as the user marks statements as faulty and non-faulty. The intuition behind the technique is that, once a statement is labeled as non-faulty, the other statements executed in the same failing test case should be considered more suspicious. Like traditional SFL approaches, and unlike , their technique requires developers to determine the correctness of individual program statements without contextual information, which has been shown to be problematic [41].

Ko and Myers proposed Whyline [5, 21], an interactive debugger that lets a developer trace incorrect variable values backwards by asking questions about how these values came to be. Whyline is similar in spirit to dynamic backward slicing—the user follows a sequence of incorrect variable values through program dependence chains to get to the fault. More recently, Lin and colleagues proposed Microbat [6], a feedback-driven debugging technique that improves on Whyline by inferring patterns in execution traces and using developer feedback to skip partial program executions, expediting the backward tracing process. Xu and colleagues proposed a technique that uses probabilistic inference for fault localization [7]. The technique builds the probabilistic model based on the DDG, associating with each DDG edge a probability that an incorrect value of the predecessor would lead to an incorrect value of the successor, and the computing the most likely explanation of the observed faulty output value. , and the three techniques described above all leverage lightweight user feedback to improve fault localization. However, in contrast to these other techniques, the queries produces are contextualized by method invocations as opposed to focused on arbitrary execution points. This feature not only lets the developer obtain relevant contextual information when answering specific queries, but also enables the technique to jump across calling contexts guided by the suspiciousness of program statements.

Whyline [5, 21] and Microbat [6] are both dependency-based debugging approaches and are also related to TESSERACT, which is partly inspired by these techniques. Different from Whyline and Microbat, TESSERACT parallelize the dynamic analysis on computing clusters and focuses on scaling the dependency-based debugging approach to handle realistic executions. Moreover, TESSERACT not only allows developers to inspect a statement instance that is reached by following dynamic dependencies, but also reproduces the full program states at the moment when the statement instance was executed.

The debugging technique proposed by Hao and colleagues [8] sets breakpoints in the faulty program using suspiciousness of program statements given by SFL. At each breakpoint, the technique asks the developer to inspect the program using a debugger to determine whether the program state has been infected by the fault. The suspiciousness of related statements is then increased or decreased by a fixed ratio based on the provided feedback. In contrast to their approach, selects for inspection a small set of suspicious data items within selected method invocations; it does not require the developers to find faulty memory locations in the entire program state. In addition, incorporates developers' feedback into the SFL algorithm, so as to dynamically update suspiciousness information. In follow-up work, Hao and colleagues proposed VIDA [103], which leverages program dependences to find statements whose suspiciousness must be updated. Compared to VIDA, asks for feedback on the input-output relations of methods, whose intended behavior tends to be well understood, rather than on individual program statements.

4.2 Program Slicing and Dynamic Dependency Analysis

To reduce the cost of debugging, Weiser proposed *program slicing* [104, 105, 106]. Given a variable v at a program point p, a *static program slice* contains the subset of statements that might affect the value of v at p in all possible program executions. To find a fault, developers do not need to consider all statements of the program, but can focus on the program slice for the observed faulty variable. Many subsequent studies proposed a variety

of program slicing techniques [107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131]

Korel and Laski first proposed dynamic program slicing [26]. Their technique identifies the statements that actually influence the slicing criterion in a specific execution by computing the dependences among the instances of statements in the execution trace and generates an executable slice of the program. Agrawal and Horgan developed the technique of computing dynamic slices on the dynamic dependence graph (DDG) [27]. The technique represents statement instances as DDG nodes and the dynamic dependences among them as edges. To compute a dynamic slice, it traverses the DDG from the slicing criterion and includes in the slice the reachable nodes. Agrawal and colleagues later improved their dynamic slicing technique by developing a more precise dynamic dependence analysis that handles unconstrained pointers [43]. The technique eliminates the imprecision caused by alias analysis. It modifies the standard dataflow analysis by representing the definition and use sets of program statements as concrete memory locations instead of abstract symbolic identifiers. These traditional dynamic dependency analysis techniques handles array accesses by making the results dependent on the dynamic reaching definitions of the base pointers (or array references), the indexes, and, in the case of array writes, the assigned data items. This approach could miss the causal relation between a incorrect data item and the fault if faulty program states propagate via incorrectly computed array indexes. My technique builds on top of these dynamic slicing techniques and improves their fault-inclusion capability in the context of software debugging by explicitly models this type of causal relation. Moreover, because of the high computational cost of constructing and storing DDGs, traditional dynamic dependency analysis techniques that work on a single machine are either limited to handle extremely short program executions or store DDGs in highly compressed format, which is slow to traverse. TESSERACT builds on top of these existing techniques and scales to realistic executions by utilizing parallelism and the power of computing clusters.

Dynamic slices do not contain the fault if the failure is caused by execution omission errors (i.e., not executing some statements that are expected to be executed). To handle this type of faults, researchers proposed relevant slicing techniques. Agrawal and colleagues first defined relevant slicing [132], which identifies the statements that could have affected the value of the slicing criterion if they had evaluated differently. The technique works by analyzing and extending the DDG, adding potential dependences between certain predicates and the data items that would be assigned different values if these predicates evaluated to alternative outcomes. Gyimóthy and colleagues proposed an efficient forward algorithm that computes relevant slices for all data items as the analysis processes each executed statement [19]. Zhang and colleagues implemented the relevant slicing algorithm for the C programming language [17]. They evaluated the technique on a set of faults in C programs and empirically showed its effectiveness in locating execution omission errors. Wang and Roychoudhury developed JSlicer [133], which computes dynamic and relevant slices on bytecode traces of Java programs. The technique uses a compressed representation to store execution traces efficiently and computes slices directly on the compressed form. Relevant slicing techniques can produce overly large slices because it uses imprecise static data dependence information. To address this problem, Zhang and colleagues proposed an approximate relevant slicing technique that is fully based on dynamic analysis. The technique works by forcing the program to take alternative branches at predicate statements and using the observed dynamic dependences in these alternative paths to approximate the potential dependences. My technique complements relevant slicing in terms of enhancing the fault-inclusion capability of dynamic slicing techniques. While relevant slicing identifies potential (control) dependences caused by dynamic choices of control-flow paths, my technique finds potential memory-address dependences caused by dynamically computed memory addresses.

CHAPTER 5
ENLIGHTENED DEBUGGING

This chapter presents ENLIGHTEN, an interactive, feedback-driven fault localization technique. This work was originally published in [28].

5.1 Technique

Figure 5.1 provides a high-level view of ENLIGHTEN and shows input (left side), output (right side), and main components of the technique (inside the box). As the figure shows, ENLIGHTEN takes as input a program and its test suite and produces as output the likely location of the fault. The fault localization process of ENLIGHTEN is iterative and user-driven, as indicated by the loop and the developer's avatar in the figure. Intuitively, at the beginning of the process, ENLIGHTEN has limited knowledge about what may be causing a failure. Each iteration, however, adds relevant debugging information to ENLIGHTEN's knowledge base, which helps eventually locating the bug. In the following, I first briefly describe the main components of the technique and then discuss them in detail.

1) The *Test Runner and Dependency Analyzer* component takes as input the faulty program and a test suite for the program and computes, for each test, a dynamic dependence graph, test results, coverage information, and a set of incorrect data values.

2) The *SFL Calculator* uses the test results and the coverage information to produce a ranked list of suspicious statements, using a traditional SFL approach.

3) The *Query Generator* takes as input the program, its test suite, and the artifacts produced by the Test Runner and Dependency Analyzer, and generates debugging queries using the SFL results. Each query consists of a method invocation, together with its inputs

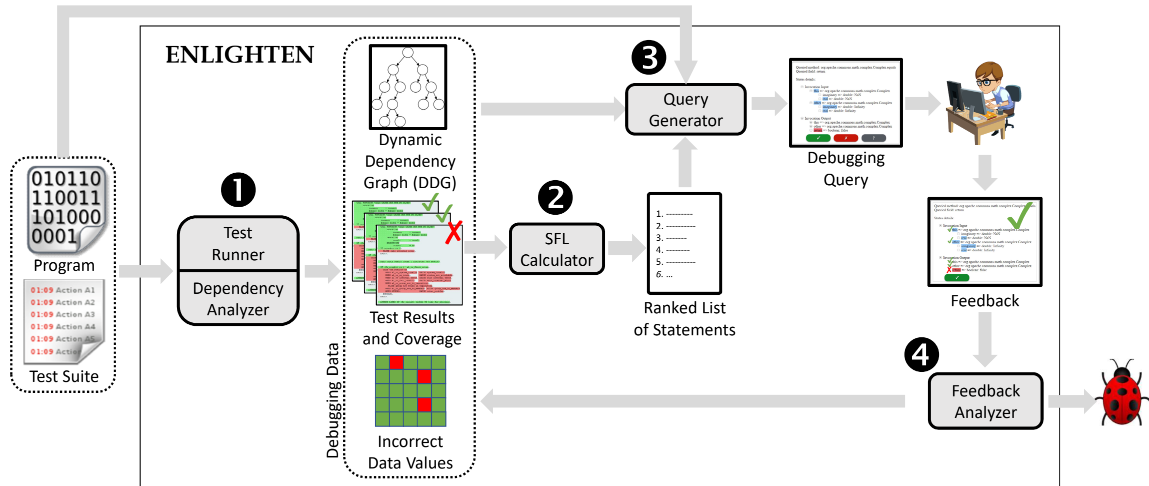

Figure 5.1: Overview of ENLIGHTEN.

(parameters plus relevant state) and outputs (including side effects), which the developer can mark as correct or incorrect.

4) The *Feedback Analyzer* takes as input the response to a debugging query. If the developer has found the bug, the process stops. Otherwise, the Feedback Analyzer updates the debugging data based on the developer feedback and performs another iteration.

Conceptually, ENLIGHTEN can operate with test suites that contain test cases triggering different faults. Multiple faults can negatively affect the initial SFL results. However, because ENLIGHTEN generates queries for a specific test case, the feedback provided by the user should overcome the noise introduced by the multiple faults. Moreover, there are several techniques that can cluster test cases that fail for similar reasons (e.g., [29, 30]) and that could be used to "specialize" the test suite before applying ENLIGHTEN.

It is also worth noting that debugging queries do *not* need to be formulated at the granularity of method invocations, and alternative partial program executions could be used instead. I choose to use method calls because methods are a fundamental abstraction developers use to reason about program semantics, and the behavior of many methods should be well understood by the developers.

My current implementation for ENLIGHTEN works for programs written in the Java

programming language and assumes that the test suite contains JUnit test cases, but the approach is general and can be adopted to other languages and testing frameworks. The prototype implementation works on deterministic executions as it requires running the test cases multiple times. To handle nondeterministic behaviors (e.g., nondeterministic system calls, different thread interleavings in multi-threading executions), the technique can be extended with deterministic record-and-replay systems [47].

5.1.1 Test Runner & Dependency Analyzer

The Test Runner is a traditional driver that takes a program and its test suite as input and produces as output the test results (pass or fail of the test oracles), coverage data, and a set of incorrect data items. This latter is a set of incorrect data items that is initialized, for each test case, with the data item associated with the corresponding failure, which is usually is the exception object created by the test oracle. (The Feedback Analyzer then adds to the set values marked as erroneous by the developers in response to queries.) In my implementation of ENLIGHTEN, which is for Java programs, a test failure can result in either an uncaught exception or a failing assertion. In these cases, the value associated with the failure is the reference to the object corresponding to the uncaught exception or the failed assertion, respectively.

The Dependency Analyzer, conversely, produces a Dynamic Dependence Graph (DDG) for every failing test in the test suite. In the DDG, nodes represent occurrences of statements in the program, whereas edges represent dynamic (data or control) dependences between these statements. As it is traditionally done, statements that contain more than one definition are split so that each node contains at most one definition [19].

5.1.2 SFL Calculator

ENLIGHTEN uses a modified version of Ochiai [31] to perform SFL. I selected Ochiai because it has been shown to perform well in practice. The specific formula I use to compute

the suspiciousness of a statement s is $susp(s)$=$a_{ef}/\sqrt{(a_{ep}+a_{ef}) \times (a_{ef}+a_{nf})}$. In the formula, a_{ef} (resp., a_{nf}) denotes the number of failing tests that covered (resp., did not cover) s. The term a_{ep} denotes the sum of the weights of the passing tests that covered s (as opposed to the number of passing tests that covered s in the original formula). The approach assigns weight 0.1 to the tests in the initial test suite and weight 1 to the virtual tests that encode the feedback provided by the user (see Section 5.1.4). The rationale for this decision is that I want the human feedback to have a high impact on the SFL results, as it relates to very focused partial executions. I picked these specific numbers so that there is an order of magnitude difference between the two. In future work, I plan to experiment with different weights and better understand their effect. In computing the formula, statements that are not executed in any test are assigned a suspiciousness score of 0.

5.1.3 Query Generator

ENLIGHTEN interacts with the developer through debugging queries, which are expressed in terms of inputs and outputs of a suspicious method invocation and are about the correctness of that invocation. I show an example debugging query in Figure 5.4 and will discuss its details in Section 5.2. The query generation process consists of (1) selecting a failing test, (2) selecting a suspicious method invocation, and (3) producing a debugging query. I now describe each of these steps.

Selecting a Failing Test

ENLIGHTEN generates debugging queries for a failing test. When multiple failing tests exist, and developers do not specify their test of choice, the technique selects the test that makes the smallest number of method calls. The rationale is that shorter traces should be easier to debug.

Selecting a Suspicious Method Invocation

Before describing how ENLIGHTEN selects suspicious method invocations, I present the concept of *value suspiciousness*. Traditional SFL techniques (e.g., [3, 31]) associate suspiciousness scores to program statements. ENLIGHTEN uses these scores to compute the suspiciousness of values defined within a dynamic method invocation (i.e., a specific runtime instance of a method execution). In the following, `slice(`$v, invoc$`)` denotes the dynamic backward slice associated with value definition v, limited to dynamic method invocation $invoc$, $v.instr$ denotes the instruction associated with definition v (i.e., the instruction that defines v), and $susp(instr)$ denotes the suspiciousness score of instruction $instr$, as computed by the SFL calculator. ENLIGHTEN computes the suspiciousness of a value definition v for a dynamic method invocation $invoc$ in two steps. First, it computes the base suspiciousness of v as $base_susp(v, invoc) = max\{susp(v'.instr)|v' \in slice(v, invoc)\}$. It then computes the actual value suspiciousness of v ($val_susp(v)$) based on whether v affects, through direct or indirect dependencies, values already known to be incorrect. Specifically, ENLIGHTEN computes $val_susp(v)$ by multiplying $base_susp(v)$ by an *amplifying factor* that is equal to either 1, if no previously labeled incorrect data item depends on v, or 1 plus the number of incorrect data items that depend on v, otherwise. Intuitively, values that affect others that developers previously labeled as incorrect are more suspicious.

To select the method invocation for the next query, ENLIGHTEN considers the output of all the invocations within the (failing) test execution being considered, where the output includes the state of the objects passed as parameters, the values of the modified global variables and objects, and the return value (or exception thrown).[1] For each such output item, ENLIGHTEN computes the corresponding value suspiciousness (i.e., the value suspiciousness of the corresponding definition). It then identifies the outputs with the highest value suspiciousness and selects the corresponding method invocation. In case of ties,

[1] ENLIGHTEN currently ignores data written through I/O operations, which could be added through additional engineering.

ENLIGHTEN prioritizes methods higher in the call chain and chooses randomly when all conditions are equal.

Producing a Debugging Query.

Conceptually, a query is a set of questions in the form "Is this value correct?". Specifically, ENLIGHTEN reports to the developer the inputs and outputs of a method call that produces the most suspicious output (see Section 5.1.3) and highlights the data values with various colors (and transparency) to indicate their relative suspiciousness. Figure 5.4 shows an example of a debugging query where the field `numElems` is classified as highly suspicious. Developers can answer positively or negatively to any number of questions in a debugging query. A positive (resp., negative) answer indicates that the developer believes the value is correct (resp., incorrect) for that specific invocation. Intuitively, labeling an output as incorrect indicates that the bug is either in the method itself or in one of the methods it calls. Note that developers can also label an input as incorrect (e.g., an unexpected $null$ value); labeling an input as incorrect tells ENLIGHTEN to ignore the current invocation and focus on methods that led to this invocation instead.

5.1.4 Feedback Analyzer

Figure 5.2 shows the algorithm for incorporating the feedback provided by developers through their answers to debugging queries (see Figure 5.1). Global variables `passingTests` and `incorrectValues`, declared at lines 1 and 2, store coverage information for passing tests and a set of known incorrect data items observed during a debugging session. ENLIGHTEN incorporates feedback by modifying these sets. At lines 5–9, the algorithm handles the case of the developer marking some values on the input as incorrect; method `getIncorrectInputValueDefs` returns the set of value definitions specified as incorrect by the developer, and the algorithm adds those values to the set `incorrectValues` and returns.

```
1  Set<Test> passingTests = ...
2  Set<Value> incorrectValues = ...
3
4  incorporateFeedback(Feedback feedback) {
5   if (feedback.isIncorrectInput()) {
6    incorrectValues.addAll(
7      feedback.getIncorrectInputValueDefs());
8    return;
9   }
10  for (Value v : feedback.getCorrectOutputValueDefs()) {
11    Test virtualTest = new Test();
12    virtualTest.setCoverage(slice(v, feedback.invoc));
13    passingTests.add(virtualTest);
14  }
15  if (feedback.hasIncorrectOutput()) {
16   response = askIfFaultFound();
17   if (response == ``yes'') return;
18   Set<Instr> directCov = feedback.invoc.getDirectCoverage();
19   if (response == ``no'') removeCoverage(directCov);
20   else { // ``I don't know''
21    Test virtualTest = new Test();
22    virtualTest.setCoverage(directCov);
23   }
24   incorrectValues.addAll(
25     feedback.getIncorrectOutputValueDefs());
26   Set<Instr> transitiveCov =
27     feedback.invoc.getTransitiveCoverage();
28   restrictSflTo(transitiveCov);
29 }}
```

Figure 5.2: Algorithm for incorporating feedback in ENLIGHTEN.

Lines 10–14 handle the case in which the developer has labeled some output values as correct. In this case, ENLIGHTEN creates a passing virtual test for each value `v` labeled as correct and updates the debugging information accordingly: function `slice` computes the dynamic backward slice from `v`, and function `setCoverage` marks the statements in the slice as covered by the newly created virtual test. Intuitively, adding passing virtual tests reduces the suspiciousness of statements related to the computation of `v`.

Lines 15–29 handle the case in which the developer has labeled some output values as incorrect, which indicates that there may be faults in the current method or in one of the methods it calls. ENLIGHTEN therefore asks the developer to check whether the fault is in the code of the current method and to provide one of three possible answers: yes, no, $I don't know$ (line 16). If the developer's answer is yes, the fault localization process ends (line 17). If the answer is no, ENLIGHTEN marks all the statements in the method as not covered (by any test), which has the effect of setting to zero the suspiciousness of all instructions in this method (line 19). (Note that this does not prevent ENLIGHTEN

from looking for the fault in methods called by this method.) Finally, if the answer is $I don't know$, ENLIGHTEN slightly decreases the suspiciousness of the current method by adding a passing virtual test whose coverage consists of the statements directly covered by the current invocation (lines 21–22).

In these two latter cases (i.e., no and $I don't know$ answers), the fault localization process then continues. As in the case of incorrect input values, ENLIGHTEN adds output values marked as incorrect to the set of known incorrect data items. Lines 26–28 then restrict the computation of SFL suspiciousness to the instructions covered, directly or indirectly, by the current invocation only.

5.2 Illustrative Example

To help illustrate the details of my approach, I introduce an example consisting of a simple faulty program. Figure 5.3 shows the code and corresponding test suite for class `BoundedStack`, which implements a stack of bounded size and which I adapted from previous work [32]. For brevity, I omitted the code that checks the capacity of the stack in method `push`. The fault is located at line 8: method `pop` should have no effect on an empty stack, but it does not check whether the stack is empty. Consequently, when the stack is empty, the method `pop` incorrectly decrements the field `numElems` denoting the stack size, which becomes negative. This class has three unit tests, and test `t3` fails with an `ArrayIndexOutOfBoundsException` when calling `bs.peek()` at line 35, after calling `bs.pop()` on an empty stack. At that point, the field `numElems` is -1, and the expression `size()-1` at line 12 evaluates to -2.

I now describe how ENLIGHTEN would support a developer in localizing the fault in this code.

First Iteration. The left table in Figure 5.5 shows the initial SFL results: line 13 is the most suspicious statement, while the actually faulty line is ranked, in the worst case, at fourth place among the eight executable statements of the program. The imprecision of

```
1  public class BoundedStack {
2
3   Integer[] elems; int numElems;
4   BoundedStack(int max) { elems = new Integer[max]; }
5
6   void push(Integer k) {// check size against capacity
7    elems[numElems++] = k; }
8   void pop() { --numElems; }
9   Integer peek() {
10   if (size() = 0)
11     return null;
12   else return elems[size() - 1]; }
13  void clear() { numElems = 0; }
14  int size() { return numElems; } ... }
15
16  @Test
17  t1() {
18    BoundedStack bs = new BoundedStack(3);
19    bs.push(3);
20    assertEquals(1, bs.size()); }
21
22  @Test
23  t2() {
24    BoundedStack bs = new BoundedStack(3);
25    bs.push(4); bs.push(5);
26    bs.pop();
27    assertEquals(4, bs.peek()); }
28
29  @Test
30  t3() {
31    BoundedStack bs = new BoundedStack(3);
32    bs.push(6);
33    bs.clear();
34    bs.pop();
35    assertEquals(null, bs.peek()); }
```

Figure 5.3: Stack and corresponding test suite example for ENLIGHTEN.

SFL is caused by the fact that line 13 happens to be invoked only in the failing test case, and it thus has a stronger correlation with test failures than the actual faulty statement.

The value stored in field `numElems`, defined during the invocation of `clear`, gets its base suspiciousness score from the suspiciousness of the definition at line 13, which is 1.0. This score is then multiplied by its amplifying factor, which is computed based on the set of incorrect data values. This set initially only contains the exception object thrown when accessing the array `elems` at line 12 in `t3`. Because this exception object has a dynamic dependence on the value stored in field `numElems`, the amplifying factor associated with that value would be 2, and the value suspiciousness for `numElems` would therefore be 2.0 (see Section 5.1.3).

In this case, the value suspiciousness of `numElems` would be the highest amongst all

values observed. ENLIGHTEN would therefore generate a debugging query for `clear`, shown in Figure 5.4, with the value of field `numElems`, marked as highly suspicious (i.e., red).

After inspecting the inputs and outputs, the developer would find that the method correctly set `numElems` to `0` and respond to the query accordingly. As a result, ENLIGHTEN would add a virtual test to the test suite reflecting the positive feedback from the developer on that output value. The coverage matrix on the right side of Figure 5.5 shows the updated rankings after this first iteration. Note that failing test cases and passing virtual test cases have weight 1, as described in Section 5.1.2.

Second Iteration. The statements at lines 8, 10, and 12 appear at the top of the ranking after the first iteration, with suspiciousness 0.95. Line 8 computes the value of `bs.numElems` in `bs.pop()`, while the execution of line 10 and 12 result in an array-out-of-bound exception in bs.peek(). The value of `bs.numElems` at the exit of `pop()` and the reference of the exception thrown by `bs.peek()` have thus a base suspiciousness of 0.95. Because the observed failure dynamically depends on both these values, their value suspiciousness is 1.90 (0.95×2). Since there are two invocations that result in the same (highest) suspiciousness value, let us assume that ENLIGHTEN randomly picks the call to function `peek` for the next query. In this case, the exception object (along with the "this" reference) would be the output of the call.

Given this query, the developer would realize that the exception is expected, as it is caused by a stack size that was already negative at the entry of the call. The developer would therefore mark field `numElems` in the input as incorrect. ENLIGHTEN would thus add the value (-1) in field `numElems` to the set of known-incorrect values, which has the effect of increasing the amplifying factor associated with all definitions that affect that value, and return (see Section 5.1.4).

Third Iteration. Due to the increase in its amplifying factor during the last iteration, the data value `bs.numElems` in `bs.pop()` becomes the single most suspicious value defi-

Test: t3	
Method: BoundedStack.clear		

Input	Output	Susp
|- elems[0]: 6	|- elems[0]: 6	0.0
|- elems[1]: 0	|- elems[1]: 0	0.0
|- elems[2]: 0	|- elems[2]: 0	0.0
|- numElems: 1	|- numElems: 0	2.0

Figure 5.4: Debugging query on the 1st iteration.

stmt.	t1	t2	t3	*susp.*
4	1	1	1	0.91
7	1	1	1	0.91
8	0	1	1	0.95
10	0	1	1	0.95
11	0	0	0	0.00
12	0	1	1	0.95
13	0	0	1	1.00
14	1	1	1	0.91
result	✓	✓	✗	-
weight	0.1	0.1	1	-

stmt.	t1	t2	t3	t4	*susp.*
4	1	1	1	0	0.91
7	1	1	1	0	0.91
8	0	1	1	0	0.95
10	0	1	1	0	0.95
11	0	0	0	0	0.00
12	0	1	1	0	0.95
13	0	0	1	1	0.71
14	1	1	1	0	0.91
result	✓	✓	✗	✓	-
weight	0.1	0.1	1	1	-

Figure 5.5: Coverage matrices before / after the 1st iteration.

nition, with a suspiciousness score of 2.85 (0.95×3). ENLIGHTEN would therefore select the invocation of `pop()` for the third query to the developer. The developer would likely and quickly understand the failure, by observing that the value of `this.numElems` is 0 at the entry of the call and -1 at its exit, and end the fault localization process.

5.3 Empirical Evaluation

I conducted two complementary studies to evaluate ENLIGHTEN: an analytical study with simulated users (Section 5.3.1) and a user study with real users (Section 5.3.2). The former let me evaluate my technique on a large number of data points and under various settings, which is typically challenging in studies involving real users. The study with real users, conversely, let me assess the usefulness of ENLIGHTEN when considering actual developers' behavior, which can only be approximated in a simulation.

5.3.1 Study with Simulated Users

In this study, I investigated different aspects of ENLIGHTEN using simulated users and a large number of faults. Specifically, I investigated the following research questions:

RQ1. How many iterations are necessary for ENLIGHTEN to localize a fault?

RQ2. What is the impact of the customized SFL formula and the amplifying factor on the effectiveness of ENLIGHTEN?

RQ3. How sensitive is ENLIGHTEN to incorrect user responses to debugging queries?

The first question evaluates the performance of ENLIGHTEN in a scenario in which the user always answers queries correctly. The second question assesses the usefulness of some key features of ENLIGHTEN. Finally, the third question evaluates how the performance of ENLIGHTEN degrades when the accuracy of the developers' responses degrades.

Experiment Setup

Benchmark Programs and Faults. As benchmarks, I used three open-source programs widely used in fault localization research: Math, Lang, and Jsoup. Math and Jsoup are available in their public repositories [33], whereas Lang is available in the Defects4J repository [34]. (Math is also part of Defects4J, but with different versions from the ones I considered.) I selected these benchmarks because they do not use features unsupported by Java PathFinder [35], which ENLIGHTEN currently leverages to build DDGs. Table 5.1 presents these programs and faults. Since each program has multiple versions, I report the number of classes, number of methods, and code size as numeric ranges. I considered two sets of faults: (1) 27 real faults, available together with the benchmarks, and (2) 1,780 mutation faults [36, 37], which I created using the mutation tool Major [38]. I discarded faults that traditional SFL ranked in a top position, as I wanted to evaluate ENLIGHTEN in the more challenging (and more common) cases in which vanilla SFL would not be useful.

Table 5.1: Benchmarks and faults considered for ENLIGHTEN.

Benchmark	*# Classes*	*# Methods*	*kLOC*	*# Faults*	
				Real	*Mutation*
Math	236 - 447	1,723 - 3,899	43 - 83	11	1,174
Lang	123 - 170	1,835 - 2,281	45 - 57	8	490
Jsoup	75 - 206	611 - 1,032	8 - 14	8	116

This led to discarding 3 of the 30 real faults available. For the mutation faults, I ran Major in its default configuration and only considered mutants killed by at least one failing test case.

Simulated Users (Automated Oracles). I used automated oracles, in lieu of real users, to answer the queries that ENLIGHTEN generated. Consider a query involving a specific invocation i of a method. To suitably classify an output value as correct or incorrect, the oracle re-runs i using the correct version of the program and compares this expected output with that of i's actual execution. To ensure that i is executed with the same input as the faulty program, the oracle starts the test execution on the faulty version and replaces the definition of the faulty class with the correct one right before invoking i, using runtime class re-definition [39].

I assume deterministic executions, so that any difference in program state between the two runs on the faulty and correct versions can only be caused by the fault. Also, for each query, the oracle only provides feedback on the most suspicious of the output values, rather than on multiple ones. Note that providing feedback on multiple values would help locate the fault in fewer iterations, and thus likely improve the performance of my technique. However, I believe that the approach I chose mirrors well the behavior of a real user, who is more likely to focus on one or at most a few output values than on all of them. I confirmed this in the user study (see Section 5.3.2).

In the simulated study, ENLIGHTEN terminates when (1) the current most suspicious data value is actually faulty (i.e., it has been produced by a faulty statement), and (2) this

value is computed directly in the current queried invocation. If these two conditions are not met within 100 iterations, ENLIGHTEN terminates the fault localization process and considers it failed.

Metrics. I used two metrics for evaluating the effectiveness of ENLIGHTEN: (1) the number of queries answered by the simulated user before finding the fault and (2) the number of distinct method invocations involved in such queries (the same invocation can become the most suspicious more than once). I consider these metrics reasonable approximations of developer effort: the former measures the number of interactions between the developer and the tool; the latter measures the number of times the developer needs to understand a new invocation (i.e., partial execution).

RQ1: How many iterations are necessary for ENLIGHTEN *to localize a fault?*

To answer RQ1, I ran ENLIGHTEN on my benchmarks and faults. I discuss the results for the real faults and those for the mutation faults separately. Table 5.2 shows the summary of the results for the real faults. Column "*Fault ID*" shows the identifier of the faults documented in the repositories from which I obtained them. Column "*IRoF*" (Initial Rank of Fault) shows the statement-level rank of the fault produced by SFL on the first iteration of a debugging session. Column "*# Invocs*" shows the number of distinct method invocations in the queries produced to locate the fault. Column "*# Queries*" shows the number of queries answered by the simulated user before finding the fault. Column "*default*" shows the results obtained with the default configuration of ENLIGHTEN, whereas the remaining columns show results obtained using alternative configurations (see Section 5.3.1).

ENLIGHTEN successfully localized 23 of the 27 (85%) faults within 10 iterations or less, and 26 of the 27 (96%) faults within 28 iterations or less. In 11 cases ENLIGHTEN required only 1 query to localize the fault, even though SFL did not rank the faulty line first. Considering all the faults in the study, the average number of iterations necessary

Table 5.2: Summary of results for real faults for ENLIGHTEN.

Benchmark	*Fault ID*	*IRoF*	*#Invocs*	*#Queries*		
				default	*w/o Wt*	*w/o AF*
Math	C_AK_1	5	2	2	2	4
	EDI_AK_1	37	2	2	2	6
	F_AK_1	36	3	3	4	3
	M_AK_1	112	8	10	22	13
	VS_AK_1	16	1	1	2	3
	CDI_AK_1	26	3	28	32	38
	CRVG_AK_1	62	6	23	19	19
	F_AK_2	9	1	1	1	5
	MU_AK_1	29	1	1	1	10
	MU_AK_4	36	3	3	4	8
	URSU_AK_1	13	1	1	1	1
Lang	b10	63	10	16	39	17
	b16	53	1	1	1	7
	b24	65	1	1	1	64
	b26	114	-	-	-	-
	b28	5	1	2	1	1
	b39	53	2	2	2	4
	b5	7	1	1	1	1
	b6	17	3	3	4	6
Jsoup	1_3_4-1	3	1	1	1	11
	1_3_4-3	73	4	4	4	-
	1_4_2-1	16	1	1	1	1
	1_5_2-2	21	2	2	2	2
	1_5_2-5	20	1	1	1	1
	1_6_1-1CR1	56	2	9	16	8
	1_6_1-1CR2	3	1	1	1	14
	1_6_3-3	36	5	5	3	6
Average		-	2.58	4.81	6.46	10.12

for localization was 4.81 ($min = 1$, $max = 28$), and the average number of invocations involved was 2.58 ($min = 1$, $max = 10$).

I manually inspected the case of Lang b26, for which ENLIGHTEN fails to locate the fault with less than 100 queries. The faulty invocation is selected for the first time on the 15th debugging query. The suspicious output of this invocation is a string that is partially incorrect. However, due to the particular way the assertion of the failing test is written, the amplifying factor for all characters in the string is the same, and the oracle fails to identify the character that is actually incorrect. In subsequent debugging queries, the same faulty invocation is selected several times, but the oracle keeps missing the incorrect character for the same reason. I conjecture that in this case a real developer would be more likely to

Table 5.3: Summary of results for mutation faults for ENLIGHTEN.

Benchmark	*#Mut.*	*# Queries* [1, 1]		[2, 10]		[11, 100]		*Not Found*	
Math	1,174	915	77.94%	215	18.31%	29	2.47%	15	1.28%
Lang	490	438	89.39%	47	9.59%	5	1.02%	0	0.00%
Jsoup	116	77	66.38%	34	29.31%	2	1.72%	3	2.59%
Total	1,780	1,430	80.34%	296	16.63%	36	2.02%	18	1.01%

spot the error in the output string and provide the right feedback, as humans tend to view strings as a whole instead of as individual characters. (The oracle is purposely weak to avoid unfairly favoring my technique and considers the string as multiple values.)

I also analyzed the correlation between *IRoF* and *# Queries* and between *IRoF* and *# Invocs*. The Pearson's correlation coefficient [40] between the number of queries and the initial rank of the fault is 0.38, which suggests a weak positive correlation. The correlation coefficient between the number of distinct invocations in the queries and the initial rank of the fault is 0.67, suggesting a moderate to strong positive correlation. Overall, the results suggest some correlation between the problem difficulty, as measured by *IRoF*, and the performance of ENLIGHTEN. However, the data also suggests that, even in cases where *IRoF* has a considerably high value, *# Queries* can be fairly low (e.g., Math.M_AK_1 and Lang.b10).

Table 5.3 shows the summary of the results for the mutation faults. Column "*# Queries [min, max]*" shows the number of mutants for which the number of queries needed to locate the corresponding fault was between the indicated min and max values. For example, only one query was necessary to locate 915 faults (i.e., mutants) in Math, whereas between two and ten queries were necessary to localize 47 faults in Lang. Overall, ENLIGHTEN successfully localized 99% of the 1,780 mutation faults, and on average, over 96% of all mutation faults were localized with at most 10 queries. ENLIGHTEN failed to localize the fault in only 1.01% of the cases. The results suggest that ENLIGHTEN works slightly better for mutation faults than for real faults, at least for the cases I considered. The reason may

be that many of the real faults are inherently more difficult to debug—a conjecture that is potentially supported by the observation that some of these faults were present in the released versions of popular libraries.

RQ2: What is the impact of the customized SFL formula and the amplifying factor on the effectiveness of ENLIGHTEN*?*

The weights used to compute statement suspiciousness and the amplifying factor used to compute value suspiciousness are two important aspects of the design of ENLIGHTEN. This research question evaluates their effectiveness. To answer RQ2, I ran ENLIGHTEN disabling each of these features separately and compared the results so obtained with those obtained using both features.

Table 5.2 shows the results for this study in the columns labeled "*# Queries*". Column "*w/o Wt*" shows the number of answers to queries that ENLIGHTEN needed to locate the fault when weights were *not* taken into account (i.e., I simply set to 1 the weights of all tests, which are used to compute the term a_{ep} of the SFL formula in Section 5.1.2). Results show that, on average, ENLIGHTEN needed 6.46 queries in this setting, compared to 4.81 queries in the default configuration, which correspond to a 34% increase. Column "*w/o AF*" shows the number of queries when the amplifying factor (AF) was ignored (see Section 5.1.3). When using this configuration, ENLIGHTEN failed to locate the fault Jsoup.1_3_4-3 and needed, on average, 10.12 queries to locate the remaining faults. This corresponds to a 110% increase over the default configuration. Note that, due to the statistical nature of ENLIGHTEN, it is possible for the configurations "*w/o Wt*" and "*w/o AF*" to perform slightly better in some cases (e.g., Math CRVG_AK_1), but these cases are rare.

I observed similar results on mutation faults, which I do not report for space reasons. For "*w/o Wt*", the success rate of locating the fault decreased by 0.5%, and the average number of queries increased by 3.7%. For "*w/o AF*", the success rate decreased by 1.8%, and the average number of queries increased by 139%.

These results indicate that the customized SFL formula and the amplifying factor both contribute to improve ENLIGHTEN's performance. The contribution of the customized SFL formula is lower compared to the contribution of the amplifying factor.

RQ3: How sensitive is ENLIGHTEN *to incorrect user responses to debugging queries?*

So far, I have assumed that developers do not make mistakes. In practice, however, they can err by labeling correct values as incorrect or vice versa. This research question investigates how sensitive is the performance of ENLIGHTEN to incorrect data items labeled as correct. (I leave to future work the investigation of the opposite case, which I consider less likely to happen.) To conduct this study, I modified the automated oracle so that it produced this type of erroneous answers with a configurable probability. Specifically, I considered error rates ranging from 5% to 30%, with 5 percentage points increments, and measured the number of queries and the number of cases in which ENLIGHTEN fails. As before, I configured the oracle to provide only one answer per query.

Table 5.4 shows, for each benchmark and for the different error rates considered, the average increase in the number of queries necessary to localize a fault over the case of an ideal oracle (i.e., a user that does not make mistakes). For example, when the erroneous answer rate is 30%, ENLIGHTEN needs, on average, 42.67% more queries to locate a fault. The results in the table show, as expected, a positive correlation between the rate of erroneous answers and the increase in the number of answers required to locate a fault. However, the results also show that ENLIGHTEN is still able to localize the fault in almost all cases. Even with 30% erroneous answers, the average success rate was higher than 99.8%.

These results show that, although the number of queries needed to localize a fault increases with the ratio of erroneous answers, ENLIGHTEN can successfully locate the fault in most cases even in the presence of (considerable amounts of) erroneous feedback.

Table 5.4: Sensitivity of ENLIGHTEN to human errors. Values indicate the percentual increase in the number of queries over an ideal oracle (i.e., a user that does not make mistakes).

Benchmark	Error rate					
	5%	*10%*	*15%*	*20%*	*25%*	*30%*
Math	5.92%	12.26%	18.82%	25.46%	32.08%	38.58%
Lang	6.08%	14.36%	21.93%	29.44%	36.74%	43.75%
Jsoup	7.63%	15.88%	24.37%	32.82%	41.02%	48.85%
Average	6.13%	13.89%	21.24%	28.58%	35.75%	42.67%

5.3.2 User Study

In addition to the study with simulated users, I conducted two actual user studies to evaluate ENLIGHTEN in a realistic scenario. The user studies involve two debugging tasks each, where each task consists of localizing and proposing a fix for a fault in a program.

Study Setup.

Benchmarks, Faults, and Participants. The software benchmarks and faults I selected are non-trivial, real faults that existed in released versions of popular software libraries written in Java. To simulate a scenario in which participants debug code with which they are familiar, I wanted software whose semantics should be well understood by a person with a computer science background. To this end, I chose code that involves either basic mathematical concepts or XML parsing. In addition, as I did for the simulated study, I selected faults for which traditional SFL techniques do not perform well (i.e., the faulty statements are not ranked among the most suspicious statements). I do so to avoid trivial cases in which SFL by itself would be enough to localize the fault.

Table 5.5 summarizes the information about the two studies I performed. For each study and each task in that study, it shows the benchmark used in the task and a concise description of the corresponding fault considered. As the table shows, the faults I used for Tasks 1, 2, and 4 were selected from a benchmark used in the simulated study, whereas the

Table 5.5: Debugging tasks for the user study for ENLIGHTEN.

User Study	*Task ID*	*Benchmark*	*Fault Description*
1	Task 1	Math	Complex number multiplication error
	Task 2	Math	Least common divisor computation error
2	Task 3	Nanoxml	XML qualified name parsing error
	Task 4	Jsoup	Absolute address construction error

fault I selected for Task 3 was used in a previous user study on SFL techniques [41]. It is worth noting that, although I used the same benchmark for Tasks 1 and 2, the parts of the program involved in the two tasks are different. In other words, completing Task 1 should not affect the participants' performance in Task 2. (Even if it had an effect, it should benefit equally participants performing Task 2 with and without ENLIGHTEN.)

I determined the difficulty levels of the subject faults based on the experimenters' assessment, the lengths of the failing executions, and the success rate and debugging time of participants during pilot studies. The pair of debugging tasks in each study are of similar difficulty, but the tasks in Study 2 are significantly harder than those in Study 1. This let me evaluate how ENLIGHTEN performs on faults at different difficulty levels.

For each of the studies, I recruited 12 participants (different for each study). The participants are graduate students enrolled in the computer science program either at Georgia Tech or at the Federal Univesity of Pernambuco. I also required the participants to (1) have at least three years of programming experience and (2) be familiar with the Java language and the Eclipse IDE.

For each study, I randomly assigned the participants to one of two groups: Group A or Group B. Participants in Group A performed Task 1 (Study 1) or Task 3 (Study 2) without ENLIGHTEN and Task 2 (Study 1) or Task 4 (Study 2) with ENLIGHTEN. Participants in Group B performed Task 2 (Study 1) or Task 4 (Study 2) without ENLIGHTEN and Task 1 (Study 1) or Task 3 (Study 2) with ENLIGHTEN. The participants not using ENLIGHTEN were allowed to use their preferred traditional debugging approach(es) (e.g., the Eclipse IDE debugger, print statements, step-by-step execution).

I used traditional debugging approaches instead of SFL as the baseline for two reasons. First, existing studies show that SFL tends to produce no measurable advantages over traditional debugging [**Wang:2015:EUI:2771783.2771797**, 41], so I do not expect user performance to improve using SFL instead of traditional debugging. Second, I believe that traditional debugging is a more objective baseline, as it relies on mature/well-accepted tools known to the participants.

I implemented ENLIGHTEN as a plugin for the Eclipse IDE and distributed the materials for the user study as a virtual machine image, so as to ensure a uniform experience across all participants. I informed the participants that I would measure their performance while debugging using two debugging approaches, without mentioning that ENLIGHTEN was my technique. Before the study began, the participants read a tutorial on the ENLIGHTEN plugin. When done with the tutorial, they performed their assigned debugging task. The time limit for each debugging task in Study 1 and Study 2 was 20 and 30 minutes, respectively.

In pilot studies, I found that the participants gave up on their tasks due to the complexity of the code involved and their lack of understanding of (some of) that code. Therefore, when performing the actual study, I allowed participants in all groups to ask questions about the semantics of a piece of code during the debugging process. This is akin to the common scenario in which the person who is performing the debugging task asks questions about the code to a developer with deeper knowledge of the software involved. I made sure to answer only general questions about what the methods were supposed to do, and I did not answer any questions about the faults being diagnosed.

Results.

Table 5.6 and Table 5.7 show the results of the two studies. In both tables, the first two columns show the ID and the corresponding group for each participant. Columns labeled "Success" indicate whether the participant correctly identified the fault in the debugging tasks ("Y") or not ("N"). Columns labeled "Time (min)" report the time spent in localizing

Table 5.6: Results for User Study 1 for ENLIGHTEN.

Participant	*Group*	*Task 1 (Traditional)*		*Task 2 (*ENLIGHTEN*)*	
		Success	*Time (min)*	*Success*	*Time (min)*
1	A	Y	17.5	Y	5.4
3	A	Y	8.9	Y	11.0
5	A	Y	8.0	Y	10.8
7	A	Y	5.5	Y	8.5
9	A	Y	18.3	Y	16.0
11	A	Y	25.1	Y	16.4
		Task 2 (Traditional)		*Task 1 (*ENLIGHTEN*)*	
		Success	*Time (min)*	*Success*	*Time (min)*
2	B	Y	19.7	Y	8.6
4	B	Y	20.1	Y	9.0
6	B	Y	12.2	Y	4.0
8	B	Y	20.8	Y	9.5
10	B	Y	18.9	Y	8.4
12	B	Y	11.0	Y	5.4
Average		100%	15.5	100%	9.4

the fault (in case of success). For both groups, the results for the task performed using traditional debugging are shown in the 3rd and 4th columns, and the results for the task performed using ENLIGHTEN are shown in the 5th and 6th columns. The last row in each table shows the average success rate and debugging time for each task and technique.

In Study 1, all participants successfully completed both of their debugging tasks. On average, participants took 15.5 minutes to complete the tasks using traditional debugging, and 9.4 minutes to complete the tasks using ENLIGHTEN. Therefore, for the tasks considered, ENLIGHTEN reduced the debugging time by 39% on average. This difference is statistically significant with a p-value less than 0.005 using a one-tailed t test.

In Study 2, participants successfully completed 58.3% of the debugging tasks using traditional debugging, and the average debugging time for these successful cases was 23.2 minutes. Conversely, the participants successfully completed all their tasks when using ENLIGHTEN, and the average time spent on each task was 9.5 minutes. In these cases, therefore, the use of ENLIGHTEN increased the success rate by 71.5% and reduced the debugging time by 59%. Also in this case, the differences for both metrics are statistically significant. The p-value of the one-tailed t-test of the success rates is 0.009, and that of the

debugging time is less than 0.001.

On average, for the tasks completed using ENLIGHTEN, participants needed 67% more queries than the perfect oracle to localize the faults, which indicates that humans do make mistakes in answering queries. However, it is worth noting that 71% of the participants needed exactly the same number of queries as the perfect oracle.

Comparing the reduction in debugging time in the two studies, the results seem to indicate that ENLIGHTEN improves developers' efficiency in debugging tasks more significantly for faults that are more difficult to diagnose, which I consider a positive result.

At the end of the user study, I asked the participants to complete a questionnaire about whether/how ENLIGHTEN helped them, as well as what other information could have been provided by the tool to make it easier to localize and understand the fault. The two advantages of ENLIGHTEN most frequently mentioned were that (1) it points developers to the likely faulty invocation in the execution, and (2) it provides detailed program state information for inspection. These two aspects roughly correspond to what I consider to be the main improvements I made in ENLIGHTEN over traditional debugging and traditional SFL techniques. The most wanted feature that ENLIGHTEN does not currently provide, according to the questionnaires, is the ability to get the context of the method invocation in the debugging query, including the call stack and the position of the current invocation in the entire execution. Several participants thought that this information would give them a better understanding of the entire debugging process and help them give feedback to debugging queries more efficiently. It would be straightforward to provide this additional information, and I am planning to do it in future work.

I also interviewed the participants about their general feeling on the debugging experience using ENLIGHTEN. Multiple participants mentioned that learning to use the ENLIGHTEN plugin in the time I allocated for the training was challenging. One participant specifically pointed out that it was difficult to change their debugging mindset from a traditional code-centric paradigm to a more data-centric one. Finally, several participants

Table 5.7: Results for User Study 2 for ENLIGHTEN.

Participant	*Group*	*Task 3 (Traditional)* *Success*	*Time (min)*	*Task 4 (*ENLIGHTEN*)* *Success*	*Time (min)*
1	A	N	-	Y	9.3
3	A	Y	21.9	Y	18.0
5	A	N	-	Y	9.6
7	A	Y	30.0	Y	5.9
9	A	Y	24.0	Y	6.0
11	A	Y	21.8	Y	9.9
		Task 4 (Traditional) *Success*	*Time (min)*	*Task 3 (*ENLIGHTEN*)* *Success*	*Time (min)*
2	B	N	-	Y	11.1
4	B	N	-	Y	7.4
6	B	Y	16.9	Y	7.3
8	B	Y	25.4	Y	11.0
10	B	Y	22.4	Y	4.1
12	B	N	-	Y	14.9
Average		58.3%	23.2	100%	9.5

reported that they spent a long time inspecting the code of the method in the query only to later discover that it was not necessary. I speculate that these feedback may indicate that people's performance using ENLIGHTEN could further improve after they get more familiar with the technique.

5.3.3 Limitations and Threats to Validity

The main limitation of my current implementation of ENLIGHTEN comes from the computation of the dynamic dependence information. Due to the enormous engineering effort required to develop a tool that implements my approach, the current dynamic dependency analyzer does not support some features of the Java standard library (e.g., certain encryption algorithms and Swing). This is an implementation-specific limitation and can be addressed with additional engineering. Another limitation, shared with many other debugging techniques that rely on dynamic slicing, is that the performance overhead of ENLIGHTEN during program execution can be significant. In the user study, however, no participant complained about the running time of ENLIGHTEN.

The major internal threat to validity for the evaluation has to do with possible faults in

my implementation of ENLIGHTEN that may invalidate the results. To address this threat, I carefully checked and unit tested my code during development. Furthermore, for the real faults in the benchmark, I manually inspected the interactions between the automated oracle and ENLIGHTEN to confirm that the sequences of debugging queries and feedback were correct.

The main external threat to validity is that the benchmarks I used might not be representative of faults in real-world scenarios and/or my results may not generalize. To mitigate this threat, I selected benchmarks that perform different types of tasks: manipulating complex data structures, performing numeric computations, and processing XML files. In addition, in the study with simulated users I evaluated ENLIGHTEN with both real faults and a large set of mutation faults, and in the study with real developers I used four different real-world faults. Another possible external threat is that the population of participants I recruited for the user study might not be representative of real developers. To mitigate this threat, I required the participants to have at least three years of programming experience.

CHAPTER 6
MORE ACCURATE DYNAMIC SLICING FOR BETTER SUPPORTING SOFTWARE DEBUGGING

This chapter presents potential memory-address dependences (PMD). Intuitively, PMDs represent the dependence relationship between an instruction s that affects the computation of a memory-address ma (e.g., by defining an array index or a pointer offset) and memory access instructions that are *not* observed to be dependent on s but could be affected by s (i.e., access the memory at ma) in a counterfactual execution of s, which computes a different value of ma. This work was originally published in [42].

6.1 Technique

Generally speaking, my approach is relevant for all languages in which variable values can be used to compute memory addresses, such as C, C++, C#, or Java. Typical examples of this situation are array indexes and offsets used for pointer arithmetic (for languages that provide direct access to memory). For ease of discussion, and without loss of generality, in the rest of the chapter I will focus on the Java language and on memory-address computations that involve the use of array indexes. The approach can be easily generalized to other languages and constructs with analogous effects.

6.1.1 Terminology and Definitions

Before describing the approach, I briefly introduce the terminology that I use in the chapter, which is partly based on the terminology originally introduced by Agrawal and De-Millo [43]. An *execution history* is the sequence of instruction instances that a program executed for a given input. A *definition* of a memory location m is an instruction instance that modifies the value of m. A *use* of a memory location m, conversely, is an instruction

instance that reads the value of m. The *use set* and *def set* of an instruction instance s contain the *memory locations* that are read and written by s, respectively. For a memory location m, $drd(m)$ denotes its *dynamic reaching definition*, which is the unique instruction instance that performed the most recent assignment to m. A *dynamic dependence graph (DDG)* is a graph (V, E) in which V is a set of vertices, each representing an instruction instance, and $E \subseteq V \times V$ is a set of edges that represent the dynamic control and data dependences between the vertices.

I can now define potential memory-address dependence.

Definition 1 *PMD: The use of a memory location m_2 in an instruction instance s_2 at a position l_2 in the execution history is potentially memory-address dependent on an earlier instruction instance s_1 at a position l_1 if*

1. *s_1 defines a memory location m_1,*
2. *m_1 is directly or indirectly (i.e., transitively) used to compute a memory address ma that identifies a memory location $m_{ma} \neq m_2$,*
3. *an instruction instance s_{def} defines m_{ma},*
4. *m_2 is not redefined between s_{def} and s_2, and*
5. *had s_1 assigned a different value to m_1, m_{ma} and m_2 would have been the same memory location.*

I call s_1 a *PMD predecessor* of s_2.

6.1.2 Motivating Examples

Figure 6.1 shows an example involving a failing test in which the faulty program state propagates through PMDs. The system under test (SUT) is method `incrementFirstInt`, which should increment the first element of the input array but has a fault at line 4; it uses

```
1  void incrementFirstInt(int[] array) {
2          // The value of the variable "index" is faulty.
3          // The correct value is 0.
4          int index = 1;
5          ++array[index];
6  }
7
8  @Test
9  void incorrectArrayWrite() {
10         int[] array = new int[2];
11         array[0] = 0;
12         array[1] = 1;
13         incrementFirstInt(array);
14         assertEquals(1, array[0]);
15 }
```

Figure 6.1: Example of PMD caused by array writes.

index value 1, instead of 0, to refer to the first array element and therefore incorrectly increments the second element of the input array. The test method fails at line 14 because of the unexpected value in `array[0]`. Using the observed incorrect value at line 14 as the slicing criterion, neither traditional nor dynamic relevant slices would contain the actual faulty statement because the statement has no observable effect on `array[0]` through explicit control or data dependences. Instead, `array[0]` is incorrect because, due to the faulty index value at line 4, line 5 modifies an incorrect memory location (i.e.,`array[index]`). The technique would handle this case by creating a PMD edge in the DDG from the node corresponding to the faulty definition of `index` at line 4 to the node corresponding to the use of the first array element at line 14. Intuitively, this PMD encodes the fact that line 5 would have assigned `array[0]` a different value had `index` at line 4 been assigned value 0.

Figure 6.2 shows a second example of a faulty program state that propagates through PMDs. The SUT method `incrementFirstElement` takes a reference-type array of `MutableInt`, which is defined on lines 1–4. Similar to the previous example, the method is faulty because of the use of a wrong index value at line 9. As a result, line 10 incorrectly reads the second element of the array, causing variable `firstElement` to point to a wrong object. As a consequence, the subsequent field assignment through `firstElement` on line 11 modifies an incorrect memory location (i.e., `array[1].value`). After executing line 11, field `array[0].value` is potentially dependent on the definition of the index at

```
1  class MutableInt {
2          int value;
3          MutableInt(int value) { this.value = value; }
4  }
5
6  void incrementFirstElement(MutableInt[] array) {
7          // The value of the variable "index" is faulty.
8          // the correct value is 0.
9          int index = 1;
10         MutableInt firstElement = array[index];
11         ++firstElement.value;
12 }
13
14 @Test
15 void incorrectArrayRead() {
16         MutableInt[] array = new MutableInt[2];
17         array[0] = new MutableInt(0);
18         array[1] = new MutableInt(1);
19         incrementFirstElement(array);
20         assertEquals(1, array[0].value);
21 }
```

Figure 6.2: Example of PMD caused by array reads.

line 9; had the index been assigned the correct value, which is 0, `array[0].value` would have been correctly modified. To represent this fact, the technique would add to the DDG a PMD edge from the node representing the (faulty) definition of `index` at line 9 to the node corresponding to the use of `array[0].value` at line 20.

6.1.3 Computing PMDs

In this section, I discuss the computation of PMDs that are caused by array reads and writes, as these are the main occurrences of this kind of dependences in the case of the Java language. However, the same approach could be extended to other languages and other constructs that result in PMDs, such as pointers and pointer arithmetic. Before discussing the specific algorithms I defined to compute PMDs for array reads and writes, I provide some preliminary definitions and high-level descriptions. The technique maintains, for each memory location m in the program, a *PMD set*, indicated as $pmd(m)$, which contains the PMD predecessors of any subsequent use of m. The technique computes and updates the PMD sets for each instruction instance in the execution history while processing the instruction instances in the order in which they are executed. The technique uses the PMD sets to create PMD edges on the DDG, as follows. For each instruction instance s in the

memory history, the technique adds to the DDG PMD edges from the nodes in the PMD sets of all the memory locations in the use set of s to the DDG node that corresponds to s.

Computing PMDs Caused by Array Writes

Conceptually, the technique processes an array assignment instruction instance `array[index]=value` by updating the PMD sets of the array elements in two steps. First, it sets $pmd($`array[index]`$)$ to the empty set because the element that was just assigned no longer depends on previously executed instruction instances. Second, it extends $pmd($`array[i]`$)$ for any other array element (i.e.,`i != index`) by adding the reaching definition of `index` because `array[i]` would have been re-assigned by the instruction instance if the index had taken the value `i`.

Updating the PMD sets of all array elements for all array assignments, however, is computationally expensive. I therefore propose an efficient algorithm to compute PMD caused by array writes based on two observations. The first observation is that each array assignment affects the PMD sets of only the elements in the array on which it operates. The second observation is that, once the PMD set for the element that is actually defined is updated, the PMD sets for all other elements are updated in the same way. As a consequence, the algorithm does not update the PMD sets individually for each array element, but instead associates the PMD information with the array object and shares the information among all elements.

Figure 6.3 shows the algorithm. In the algorithm, the statement "`var` $\leftarrow \langle e_1, e_2, \ldots, e_n \rangle$" creates a tuple that contains members e_1 through e_n and assigns it to the variable `var`. I use the expressions `var.first`, `var.second`, and so forth to represent accesses to the members of the tuple at the corresponding positions. (I use the same terminology in Section 6.1.3.)

For each array object, the algorithm stores the history of array writes as a list of write records. A *write record* is a pair (i.e., 2-tuple) that contains the dynamic reaching defi-

```
1:  procedure PROCESSARRAYWRITE(pArray, pIndex)
2:      array ← ∗pArray
3:      index ← ∗pIndex
4:      prevWr ← lastWr(array)
5:      newWr ← ⟨ drd(pIndex), prevWr ⟩
6:      lastWr(array) ← newWr
7:      lastWr(array[index]) ← newWr
8:  end procedure
9:
10: function DEFERREDPMD(array, index)
11:     deferredPmdSet ← ⟨ lastWr(array), lastWr(array[index]) ⟩
12:     return deferredPmdSet
13: end function
14:
15: function COMPUTEPMD(deferredPmdSet)
16:     pmdSet ← ∅
17:     startWr ← deferredPmdSet.first
18:     endWrExcl ← deferredPmdSet.second
19:     currentWr ← startWr
20:     while currentWr != endWrExcl do
21:         pmdSet ← pmdSet ∪ { currentWr.first }
22:         currentWr ← currentWr.second
23:     end while
24:     return pmdSet
25: end function
```

Figure 6.3: Algorithm for computing PMDs due to array writes.

nition of the index of an array assignment as the first member, and a link to the previous write record as the second member. I use `lastWr(array)` to denote the head of the array-write history (i.e., the most recent write record for the array). For each array element `array[index]`, the algorithm also keeps track of the most recent write record of the element, `lastWr(array[index])`.

The algorithm extends standard dynamic dependence analysis. Procedure `ProcessArrayWrite` is called by the dynamic analysis engine for each array assignment instruction instance to update the array-write history. Line 5 creates the new write record for the assignment, `newWr`. Line 6 sets `newWr` as the head node of the write history of the current array. Line 7 sets `newWr` as the most recent write record of the assigned element.

During DDG construction, when a PMD set of an array element `array[index]` is

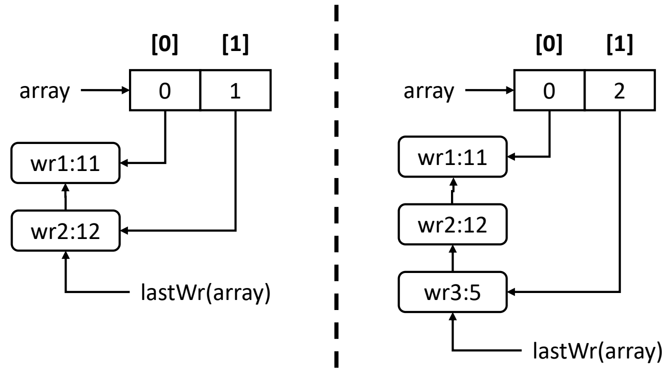

Figure 6.4: Computing PMDs for the first motivating example.

requested, the dynamic analysis engine calls function `DeferredPmd`, which takes the array reference and the index as input and returns a *deferred PMD set*—a set that contains sufficient information to compute PMDs at any later position in the execution history. Specifically, this set consists of a pair that contains the most recent write record of the array (`lastWr(array)`) as the first member, and the most recent write record of the element (`lastWr(array[index])`) as the second member (line 11). The write records in the array-write history from the first member (inclusively) to the second member (exclusively) correspond to the array assignments that should result in updates of the PMD set of `array[index]`. Based on this idea, function `ComputePmd` computes the actual potential dependences from a deferred PMD set. Starting from the first member of the deferred PMD set, the function (1) traverses the list containing the array-write history, (2) adds the dynamic reaching definitions of the indices stored in the visited write records into the initially empty collection `pmdSet` until it reaches the second member of the deferred PMD set, and (3) returns `pmdSet`.

Procedure `ProcessArrayWrite` and function `DeferredPmd` are both used during DDG construction and run in constant time for each array assignment instruction instance. Function `ComputePmd` is used only during the traversal of the DDG, and its total running time in a traversal is proportional to the number of visited PMD edges.

Figure 6.4 demonstrates how the algorithm works on the motivating example in Fig-

ure 6.1. The two diagrams separated by the vertical dashed line show the histories of array writes before and after executing line 5. I focus on this line of code because it creates the PMD edge that is related to the failure. The rectangles in the figure represent array buckets. The number inside each rectangle shows the concrete value of the bucket, while the number enclosed in brackets above the rectangle indicates its index. The rounded rectangles represent write records. The label inside each rounded rectangle shows the name of the write record and the line number of the code that creates it, separated by a colon. The arrows between the rounded rectangles show the links between the write records. For simplicity, I omit the dynamic reaching definitions of the indices in the write records from the diagrams.

The diagram on the left shows the array write history before executing line 5. It contains two records: `wr1`, created by line 11, and `wr2`, created by line 12. The most recent assignment to the array element `array[0]` is on line 11, indicated by the arrow from `array[0]` to `wr1`. Similarly, the most recent write record of `array[1]` is `wr2`, which is also the head of the write history of the whole array `lastWr(array)`. The diagram on the right shows the modified array write history after executing line 5. The technique creates a new write record `wr3` and inserts it at the head of the array write history. Since the statement modifies `array[1]`, the technique also updates `lastWr(array[1])` to point to `wr3`. To compute the PMD set of `array[0]`, the technique (1) traverses the array write history from `wr3`, the most recent write record, to `wr1`, the last assignment to the array bucket, and (2) identifies the reaching definitions of the array indices in `wr3` and `wr2` as the potential memory-address dependences of `array[0]`. In particular, the array index in `wr3` has the incorrect value that leads to the failure.

Computing PMD Caused by Array Reads

Reading an element from a reference-type array leads to PMDs if the reference read is subsequently used, either directly or through additional field/array-element accesses, to modify a memory location. For simplicity, hereafter I refer to fields and array elements

uniformly as *members* of their containing objects, and use the term `oRef.mName` to denote access to a member of `oRef`, where `mName` is either a field name or an array index.

Consider a member-write instruction instance `oRef.mName=value` at an execution point l_2, where the object reference `oRef` is retrieved from an array `refArray` through the following sequence of member-read instruction instances:

r_0=refArray[index]; r_1=r_0.mName$_1$;

r_2=r_1.mName$_2$; ......; oRef=r_{n-1}.mName$_n$;

The instruction instances in the sequence are in the same relative order as they appear in the execution history but are not necessarily executed contiguosly. I denote the execution position of the first member-access instruction instance in the sequence, which is the array read `refArray[index]`, as l_1. If the index at l_1 had a different value `i`, where `i!=index`, the member-write instruction instance at l_2 would use a potentially different object reference `oRef'`, and as a result, the program would write a potentially different memory location `oRef'.mName`. After executing the member-write instruction, the technique adds a PMD edge from the dynamic reaching definition of `index` at l_1 to the corresponding alternative memory location `oRef'.mName` for each valid `i` ($0 \leq$ i<refArray.length $\wedge$ i!=index).[1]

To compute the alternative object reference `oRef'` at l_2, the technique replaces the index used in the array-read instruction instance at l_1 with the alternative value `i` and re-evaluates the sequence of member reads that retrieved `oRef` from the array `refArray` in the actual execution. To do so, the technique encodes the member-read sequence as an access path and associates it with the object reference `oRef`. An *access path* is a pair that contains the starting object reference `r` and a list of *member-read records* that store the names of the accessed members and the timestamps of the read instruction instances. (As a clock, the technique uses an integer counter that it increments for each executed instruction.) Intuitively, the access path represents how the associated object reference is retrieved from `r`. The technique creates only access paths that start with array references. The access path

[1]To simplify the discussion, and when it is not ambiguous, I describe PMDs by using memory locations instead of the corresponding node representing statements that use or define these memory locations.

associated with oRef has refArray as the first element, and the list [⟨index, t_0⟩, ⟨$mName_1$, t_1⟩, ⟨$mName_2$, t_2⟩, ..., ⟨$mName_n$, t_n⟩] as the second element, where the elements t_0 through t_n are the timestamps of these member-read instruction instances.

When the technique computes the alternative object references, it re-evaluates the member-read instruction represented by a record ⟨$mName_m$, t_m⟩ as if the instructions therein were executed at time t_m. To do so, it keeps track of the historical values that reference-type members have taken during the execution for all objects. Specifically, it associates with each reference-type member a list of pairs ⟨t_k, r_k⟩, which indicate that an instruction instance assigned the value r_k to the member at time t_k. The technique sorts the list of historical values by timestamp, so that it can efficiently look up the value for a member at any specified execution point. Given a timestamp t_x, the technique finds the pair ⟨t_y, r_y⟩ that has the largest timestamp smaller than or equal to t_x by using binary search and returns r_y as the value of the member at the time t_x.

Figure 6.5 shows the algorithm for computing PMDs caused by array reads. In the algorithm, the notation "$[e_1, e_2, \ldots, e_n]$" indicates the creation of a list of elements e_1 through e_n. Functions head and tail are standard list manipulation functions.

Procedure ProcessRefArrayRead is called by the dynamic analysis engine for each read instruction instance of reference-type arrays. In addition to the memory locations of the array reference and the index, the procedure also takes as input the memory location of the resulting object reference of the read instruction instance. Line 4 creates an array-read record, which contains the index, the current time returned by function currentTick(), and the dynamic reaching definition of the index. Lines 5 and 6 create a new access path that contains only the array-read record. Line 7 associates the new access path with the resulting reference of the array-read instruction instance. Each object reference has a set of associated access paths, returned by function apSet, because it might have been retrieved from multiple arrays.

Procedure ProcessRefMemberRead is called for each read instruction instance of

reference-type members, including elements of reference-type arrays. The procedure takes as input the memory location of the operand object reference of the read instruction instance, the accessed member name, and the memory location of the resulting object reference, and suitably updates the set of access paths of the resulting reference. Line 11 creates a member-read record, which contains the name of the accessed member and the current time. For each of the access paths associated with the operand reference, line 13 calls function `extPath` to create a new access path that extends the existing path with the new record, and line 14 associates the new access path with the resulting reference. To reduce the space required to store access paths, the technique represents them as reversed linked lists that share common prefixes.

When the program writes a member of any object, the dynamic analysis engine calls procedure `ProcessMemberWrite` to update the set of historical values of the member and the PMD information. The procedure takes as input the memory location of the operand object reference, the name of the modified member, and the new value assigned to the member. Line 20 clears the PMD set of the member that is actually modified, as the technique also does for array writes. If the current member has a reference type, line 22 appends the time and the assigned value to its historical values, `memberHist`. For each access paths associated with the object reference, the algorithm gets the dynamic reaching definition of the index used by the initial array-read instruction instance from the access path (lines 25–27) and adds it to the PMD sets of the alternative memory locations that could have been written had the index been assigned different values (lines 28–29). Function `GetAltObj` (line 28) takes an access path and returns the set of alternative object references by replacing the index in its first array-read record with alternative values and re-evaluating the path.

I demonstrate how the technique works on the motivating example of Figure 6.2. I focus on lines 10–11 of the example, as PMDs caused by array reads involve only these two lines. After executing line 10, the technique creates an access path $\langle$`array, ["1"]`$\rangle$,

```
1:  procedure PROCESSREFARRAYREAD(pArray, pIndex, pRstRef)
2:      array ← ∗pArray
3:      index ← ∗pIndex
4:      arrayRr ← ⟨ index, currentTick(), drd(pIndex) ⟩
5:      mReads ← [ arrayRr ]
6:      newAccessPath ← ⟨ array, mReads ⟩
7:      apSet(pRstRef) ← apSet(pRstRef) ∪ { newAccessPath }
8:  end procedure
9:
10: procedure PROCESSREFMEMBERREAD(pObjRef, mName, pRstRef)
11:     memberRr ← ⟨ mName, currentTick() ⟩
12:     for accessPath ∈ apSet(pObjRef) do
13:         extendedPath ← extPath(accessPath, memberRr)
14:         apSet(pRstRef) ← apSet(pRstRef) ∪ { extendedPath }
15:     end for
16: end procedure
17:
18: procedure PROCESSMEMBERWRITE(pObjRef, mName, mValue)
19:     objRef ← ∗pObjRef
20:     pmd(objRef.mName) ← ∅
21:     if isRefMember(objRef.mName) then
22:         memberHist(objRef.mName).append(⟨ currentTick(), mValue ⟩)
23:     end if
24:     for accessPath in apSet(pObjRef) do
25:         mReads ← accessPath.second
26:         arrayRr ← head(mReads)
27:         indexDrd ← arrayRr.third
28:         for altObj ∈ GetAltObj(accessPath) do
29:             pmd(altObj.mName) ← pmd(altObj.mName) ∪ { indexDrd }
30:         end for
31:     end for
32: end procedure
```

Figure 6.5: Algorithm for computing PMDs due to array reads.

which starts with the reference of the array and has a single array-read record (representing the reading of the element at index 1). It then associates the access path with the resulting reference `firstElement`, which points to the `MutableInt` object at `array[1]`. (For conciseness, I omit the timestamp from the member-read records, as using the technique on this example does not require restoring historical program states.) When line 11 writes memory through reference `firstElement`, the technique computes the alternative possible values of `firstElement` by using the access path associated with it. Specifically, the algorithm checks the size of `array` and finds that the index of the array-read record in the access path has one alternative value 0. By re-evaluating the array-read instruction with value 0 as the index, the technique determines that, in this alternative scenario, `firstElement` points to the `MutableInt` at `array[0]`. Therefore, it adds a PMD edge from the dynamic reaching definition of the index in the array-read record to the memory location `array[0].value`.

6.2 Empirical Evaluation

To evaluate the cost and benefit of considering PMDs in dynamic-slicing-based software debugging, I implemented PMD slicer, a tool that considers dynamic and potential memory-address dependences, and compared it with a traditional dynamic slicer. I investigated the following three research questions:

RQ1: Do PMD slices include (real-world) faults that traditional dynamic slices would miss?

RQ2: How much larger are PMD slices compared to traditional dynamic slices?

RQ3: What is the computational cost of PMD slicer compared to a traditional dynamic slicer?

6.2.1 Experiment Setup

As a baseline technique for the evaluation, I selected a traditional dynamic slicer that I used in my previous work on debugging [28] (BaselineSlicer, hereafter). I built PMD slicer by

Table 6.1: Benchmark programs for PMD.

Program	*# Classes*	*# LoC*	*# Faults*	*# Failing Tests*
Chart	582–681	174K–210K	21	47
Closure	912–1608	88K–154K	169	529
Lang	119–170	43K–57K	57	101
Math	146–1041	26K–169K	92	130
Time	226–232	60K–63K	25	73
Total	-	-	364	880

extending BaselineSlicer, so as to minimize the risk that the differences measured in the evaluation are caused by implementation differences. Both slicers work by first building an in-memory DDG for the execution considered and then computing slices by traversing the DDG backward from the slicing criterion. I ran the experiments on a machine with an Intel Core i7-4770 3.4GHz CPU and 32GB DDR3 of memory and allocated 16GB maximum heap space to each slicer process. I did not compare the technique with relevant slicing in the empirical evaluation because I could not obtain an implementation of relevant slicing. I nevertheless provide a conceptual comparison in Chapter 4.

For the evaluation, I used a benchmark of 364 real-world faults in 5 open source Java programs from the Defects4J database [34]. I selected Defects4J because it provides, for each fault, the faulty and fixed program versions, and fault-revealing JUnit test cases. This lets me determine in an automated way whether the slices PMD slicer computed contained faulty statements. I did not include into the benchmark the Mockito program in Defects4J because it extensively uses the Java reflection API [44], which the current implementation of PMD slicer does not fully support. Table 6.1 shows relevant information about the programs in the benchmark.

In the table, columns "# Classes" and "# LoC" show the number of classes and the number of lines of source code in the benchmark programs, respectively. Because each program has multiple versions, these numbers are reported as ranges. Columns "# Faults" and "# Failing Tests" show, for each program and in total, the number of faults I considered

and the number of failing test cases that reveal these faults.

Note that I had to exclude a small fraction ($<10\%$) of the faults (and the corresponding fault-revealing tests) in the benchmark programs for various reasons. First, I used a modified version of Java PathFinder (JPF) [35, 45] to build the DDG within the slicers. Although I extended JPF to support most of the functionality of the standard Java library, the extended JPF still does not fully support GUI-related classes, reading and writing compressed files, locales, and reflection. I therefore excluded the tests that use these unsupported features. Second, to constrain the total running time for the experiment, given the large set of tests in the benchmark, I excluded tests that took longer than one minute to run on vanilla JPF (i.e., on JPF with all the dynamic analyses disabled). Finally, I excluded the fault-revealing tests that either did not result in failures or failed during the start-up phase (before entering the actual test method). It is worth noting that these issues are not related to my approach, but rather to its implementation and the experiment infrastructure I used.

To show the effects of considering PMDs caused by array writes and reads separately, I ran PMD slicer in two configurations: (1) the "write-only" configuration considers PMDs caused by array writes only, and (2) the "read-write" configuration is the full technique, which analyzes both types of PMDs. In the discussion of the results, I denote the execution of a test t on program P as $P(t)$. Additionally, I refer to the slices produced by the tool as *PMD slices* and to the slices produceed by BaselineSlicer simply as *dynamic slices*.

6.2.2 RQ1: Do PMD slices include (real-world) faults that traditional dynamic slices would miss?

To answer RQ1, I applied both BaselineSlicer and PMD slicer to each failing execution and assessed in how many cases the faults were not in the dynamic slices, but were in the PMD slices. Specifically, I identified the incorrect data items at the end of $P(t)$ (i.e., at the exit of the test method), used each of them as the slicing criterion, and checked whether the fault was included in the PMD and dynamic slices. To compute the set of incorrect

Table 6.2: Summary of cases that the faults are not in dynamic slices but are in PMD slices.

Program	*Write-only*			*Read-write*		
	VF	RP	%SC	VF	RP	%SC
Chart	47	0	-	46	0	-
Closure	529	58 (10.96%)	3.59%	254	48 (18.90%)	16.06%
Lang	101	1 (0.99%)	14.29%	101	3 (2.97%)	5.47%
Math	130	5 (3.85%)	13.82%	130	6 (4.62%)	14.85%
Time	73	2 (2.74%)	1.79%	71	1 (1.41%)	1.79%
Total	880	66 (7.50%)	4.47%	602	58 (9.63%)	15.14%

data items, I executed t on the fixed version of the program P' and compared the program states at the end of $P(t)$ with those of $P'(t)$ at the corresponding execution position. I determined the equality of scalar data items by comparing their actual values, and that of reference-type data items by comparing the objects that they referenced. I made sure the differences between the executions on P and P' in the data items that I used as slicing criteria were only caused by the fault by removing nondeterminism as follows. One main source of nondeterminism that extensively affected the tests in the benchmarks was object identity hash codes [46], which in particular may determine the positions of elements in hash maps. To remove this nondeterminism source, I changed the VM configuration so that it returned a constant value as the identity hash codes for all objects. Furthermore, I removed other sources of nondeterminism by excluding from the slicing criteria the data items that had non-unique values in multiple executions of each test.

Both BaselineSlicer and the PMD slicer produce slices that contain executed (dynamic) instruction instances. I determined whether a slice contained a fault that was caused by faulty program statements by simply checking whether the slice contained an executed instance of any of the faulty statements.

Table 6.2 reports the summary of the experiment results for each benchmark program and in total. The sections "Write-only" and "Read-write" show the results of the corresponding configurations of PMD slicer. Under each configuration, column "VF" (Verified

Failures) shows the total number of fault-revealing tests on which the PMD slicer ran successfully. For the write-only configuration, this is the same data I show in the column "# Failing Tests" in Table 6.1. For the read-write configuration, I have fewer successful cases because tracing PMDs caused by array reads for all data items is expensive, and I excluded the tests on which the running time of PMD slicer exceeds ten-minutes. (I plan to compute results for these additional tests in future work.) Column "RP" (Require PMD) reports the number and percentage of the tests in which PMD slicer, compared to BaselineSlicer, shows improved fault-inclusion capability. Specifically, each of these tests has a non-empty set D of post-execution incorrect data items such that the PMD slices for the data items in D contain the fault while the corresponding dynamic slices do not. For these tests, to show how likely the choice of a specific slicing criterion would require the developer to use PMD slices to locate the fault, in column "%SC" (% of Slicing Criteria) I report the ratio of the number of the data items in D to the total number of incorrect data items (except for uninitialized object members).

Overall, for the 880 fault-revealing test cases considered, PMD slicer in the write-only configuration located faults that could be missed by BaselineSlicer in 66 tests, which is 7.5% of all tests considered. Among these 66 tests, PMDs caused by array writes must be considered to locate the faults for, on average, 4.47% of the post-execution incorrect data items.

Under the read-write configuration, PMD slicer completed within the ten-minute time limit for 602 ($\sim$70%) of the 880 tests. The number of tests in which the full PMD slicer exhibits improved fault-inclusion capability is 58, which is 9.63% of all the tests successfully analyzed. 43 of these tests overlap with the tests reported in the "RP" column for the write-only configuration, and the other 15 tests contain the cases that require considering PMDs caused by array reads to locate the fault. In these 58 tests, the average percentage of post-execution incorrect data items that require using PMD slices to locate the faults is 15.14%. It is worth noting that a PMD slice computed with the read-write configuration is

Table 6.3: Size comparison for PMD slices.

Program	*Baseline*		*Write-only*				*Read-write*			
	SI	DI	SI	SI Inc.	DI	DI Inc.	SI	SI Inc.	DI	DI Inc.
Chart	109	203	110	0.5%	204	0.5%	110	0.5%	204	0.5%
Closure	3028	26476	3209	5.9%	27358	4.3%	3245	7.5%	27738	6.8%
Lang	82	434	92	4.7%	567	8.0%	92	4.7%	567	8.0%
Math	597	107068	726	12.4%	113230	18.2%	734	12.7%	118222	19.5%
Time	1789	9114	1820	4.1%	9302	13.8%	1821	4.1%	9306	13.9%
Overall	1675	34864	1787	6.5%	36568	8.7%	1804	7.2%	37767	10.1%

always a super set of the corresponding PMD slice computed with the write-only configuration. The number of tests reported in column "RP" in the write-only section being larger than the corresponding number in the read-write section is only due to the fact that some tests were excluded in the latter case.

The percentages of failures reported in the "RP" columns vary across the programs in the benchmark. The Closure program has a significantly larger percentage of tests on which PMD slices show improved fault-inclusion capability. I spot-checked the tests in the benchmark and found that, since the programs Chart, Lang, Math, and Time mainly consist of library methods, and their tests usually target a single method, many of these tests do not involve any array operation in the application code.

Answering RQ1: The results show that considering PMDs can be useful for debugging, as about 10% of the real-world failures in the benchmark resulted in faulty data items for which traditional dynamic slices did not contain the faults, while PMD slices did.

6.2.3 RQ2: How much larger are PMD slices compared to traditional dynamic slices?

One important factor that affects the effectiveness of using dynamic slicing for software debugging is slice sizes. Smaller slices provide more precise candidate sets of faulty statements, and thus enable developers to localize faults more efficiently and effectively. Since

Table 6.4: Running time comparison for PMD slicer.

Program	*Baseline*		*Write-only*				*Read-write*			
	GT	ST	GT	GT Inc.	ST	ST Inc.	GT	GT Inc.	ST	ST Inc.
Chart	965	0	1035	8.9%	0	-	1208	26.5%	0	-
Closure	2357	417	2810	19.0%	463	14.2%	13085	408.8%	478	27.1%
Lang	922	3	1066	10.0%	6	100.6%	1365	43.2%	9	168.8%
Math	3218	100	3450	13.1%	123	59.6%	4111	47.5%	281	74.9%
Time	1081	10	1190	10.1%	10	6.8%	3793	252.9%	11	13.9%
Overall	2048	205	2333	14.5%	230	24.8%	7226	224.7%	270	41.6%

considering PMDs adds more dependence edges to the DDG, a PMD slice is usually larger than the dynamic slice for the same slicing criterion. RQ2 quantitatively measures the average increase in slice size. For each test, I computed the dynamic and PMD slices for all incorrect data items in the post-execution program states and computed the differences in the number of both dynamic instruction instances and static instructions in each pair of corresponding dynamic and PMD slices.

I report the results for each benchmark program and the overall average in Section 6.2.3. Sections “Baseline”, “Write-only”, and “Read-write” show the results for BaselineSlicer, PMD slicer with the write-only configuration, and PMD slicer with the read-write configuration, respectively. Columns “SI” (Static Instruction) and “DI” (Dynamic Instruction) in each section show the average slice sizes in terms of static instructions and dynamic instruction instances, respectively. I included in the slices only the instructions in the system under test (SUT). I did not consider instructions in the libraries used by the SUT so as to mirror typical debugging scenarios, in which developers look for faults in their own code. Columns “SI Inc.” (Static Instruction Increase) and “DI Inc.” (Dynamic Instruction Increase) under the “Write-only” and “Read-write” sections show the average relative increases in slice sizes under the corresponding configurations, compared to traditional dynamic slices.

For the benchmarks, the average size of the baseline dynamic slices is 1,675 in terms

of number of static instructions, and 34,864 in terms of number of dynamic instruction instances. Under the write-only configuration, the average number of static instructions and dynamic instruction instances in a PMD slice are 1,787 and 36,568, which correspond to an average increase of 6.5% and 8.7% with respect to the corresponding dynamic slices, respectively. Similarly, under the read-write configuration, the average number of static instructions and dynamic instruction instances in a PMD slice are 1,804 and 37,767, which correspond to an average increase of 7.2% and 10.1%, respectively.

Answering RQ2: The results show that, compared to the corresponding traditional dynamic slices, PMD slices are only moderately larger.

6.2.4 RQ3: What is the computational cost of PMD slicer compared to a traditional dynamic slicer?

I answer RQ3 by comparing PMD slicer and BaselineSlicer in terms of the time necessary to (1) build DDGs for each test and (2) compute each slice on the generated DDGs. To compute the time spent by a slicer to build the DDG for a test, I ran the slicer on the test and measured the time t elpased from the start of the execution to when the graph was fully built. A significant part of t was used by the underlying JPF virtual machine to execute the program and is specific to my tools. To make the comparison result generalizable to different dynamic dependence analysis implementations, I also measured the time t' needed by a JPF instance with no dynamic analysis to execute the tests, and computed the graph building time as $t - t'$. To reduce the effect of data caching on running time, I measured the elapsed time of each test under each configuration for multiple times consecutively, discarded the time computed for the first run, and took the average of the subsequent runs. After computing the time to compute DDGs, I measured the time to compute slices for each incorrect data item in the post-execution program state of each test. I then computed the average increase in the time used by PMD slicer compared to BaselineSlicer.

Table 6.4 reports the results of this study. Similar to Table 6.3, sections "Baseline", "Write-only" and "Read-write" show the results of BaselineSlicer and PMD slicer with corresponding configurations. Under each section, column "GT" (Graph-building Time) shows the average time for building DDGs, and column "ST" (Slicing Time) shows the average time for computing each individual slice, both in milliseconds. In both the "Write-only" and "Read-write" sections, columns "GT Inc." (Graph-building Time Increase) report the average increase in graph building time, and columns "ST Inc." (Slicing Time Increase) report the average increase in the time of computing individual slices.

The average time necessary for BaselineSlicer to construct a DDG for the benchmarks was 2,048ms. Under the write-only configuration, the average graph construction time for PMD slicer was 2,333ms, which corresponds to an average increase in graph construction time of 14.5%. Since the algorithm in Figure 6.3 processes array writes in constant time, the time increase is mostly linear. Analyzing PMDs caused by array reads, however, can impose a significant runtime overhead. Under the read-write configuration, in fact, PMD slicer spent on average 7,226ms for constructing a DDG. Compared to the baseline, the average time increase is therefore 224.7%. The runtime overhead is especially high for long executions that use large arrays. One reason for the long DDG construction time is that, in this study, I analyzed PMDs caused by array reads for all data items. In a real-world debugging scenario, a possible way of reducing the high runtime overhead would be to compute PMDs caused by array reads only for specific data items (e.g., data items of interest for the developer or likely to be faulty based on some criterion).

The average time necessary for BaselineSlicer to compute a slice, given a DDG, was 205ms. Under the write-only (resp., read-only) configuration, PMD slicer took on average 230ms (resp., 270ms) to compute a slice, which corresponds to an average time increase of 24.8% (resp., 41.6%). For program Chart, the time for computing slices was shorter than the precision of the time utility I used to measure elapsed time. I therefore report 0 in the "ST" columns for this program and do not compute the time increases.

Answering RQ3: The results show that my technique can compute DDGs in the write-only configuration, and PMD slices in both configurations, with a computational cost between 15% and 40% higher than that of a traditional dynamic slicer. Computing DDGs that account for PMDs caused by array reads, however, can be extremely expensive. In future work, I will investigate approximate algorithms that can improve the efficiency of the approach.

6.2.5 Limitations and Threats to Validity

As the first work of PMD, this work prioritizes precision over performance. One major limitation of my current implementation is the high performance overhead of computing the PMDs caused by array reads. Another limitation is that the technique currently considers only the alternative memory locations where only one array index takes different values in an access path. This might lead to missing a fault that affects multiple index values in a single access path. For programming languages that allows direct access to the memory (i.e., C, C++), the technique does not consider PMDs that are caused by using a numerical value as the memory address because doing so would include all memory locations as the alternative locations.

The main internal threat is that my implementation could have faults. To mitigate this threat, I carefully tested my code during development and manually inspected the output of the tool. In particular, I spot-checked the cases that are reported by the tool as requiring PMD slices to localize the faults and confirmed their validity by manually debugging the failing test and checking how the faulty program states propagate.

The main external threat is that the faults in my benchmark might not be representative, and the results might not generalize. To mitigate this threat, I used a large benchmark of real-world faults in the Defects4J database, which was also widely used in previous studies of debugging techniques.

CHAPTER 7
TESSERACT: SCALABLE DEPENDENCY-BASED INTERACTIVE DEBUGGING

One common approach in software debugging is to explore program executions by following dynamic dependencies. Although effective, this approach is currently limited to short program executions because of its high running time and memory overhead. To address this limitation, this chapter presents TESSERACT, a technique that improves existing dependency-based debugging techniques by 1) using a low-overhead record-replay system to record and reproduce the failing execution; 2) decomposing the recorded execution into small time slices called epochs; 3) analyzing the epochs in parallel on a computing cluster; 4) using a lightweight analysis to select the epochs that are actually needed for debugging; and 5) running the expensive dynamic analysis on only those needed epochs, on-demand.

7.1 Technique

Figure 7.1 shows an overview of TESSERACT. The technique has four main components: the *Recorder*, the *Partitioner*, the *Comprehensive Analyzer*, and the *Inter-epoch Analyzer*. It supports a typical debugging process as follows. Given a program and its fault-revealing input, the Recorder stores complete information that is necessary to recreate the failing execution deterministically. The Partitioner splits the recorded execution into epochs by utilizing the replay system's capability of selectively turning on instrumentation for the replayed execution and divides the execution into time slices that are roughly equal in execution time during replay. The ability of analyzing each epoch separately allows the technique to parallelize the traditionally highly sequential computation of dynamic dependencies. The developer starts debugging the execution by first inspecting the last epoch e_{last}, which ends in the observed incorrect program behavior. To support that, the Comprehensive Analyzer

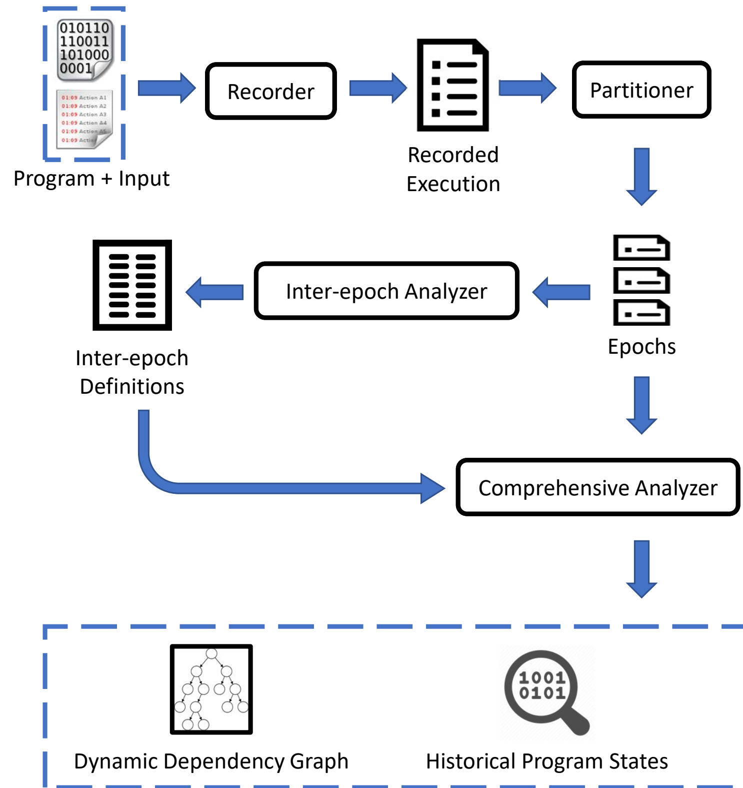

Figure 7.1: Overview of TESSERACT.

analyzes e_{last} and produces two sets of information — the dynamic dependency graph (DDG), which enables the efficient navigation of the execution by tracing dynamic dependencies, and the historical program states, which support fast rewinding of the program state to previous points in time. As the developer traces the dynamic dependencies in the backward direction, the tool may reach a dependency node where one of its predecessors is defined in a previous epoch and thus missing from the current DDG. To proceed with the debugging process, TESSERACT needs to know the specific epoch e' that defines the missing predecessor. This information is provided by the Inter-epoch Analyzer. Before the interactive debugging session starts and during the time when the developer is inspecting the last epoch, the Inter-epoch Analyzer runs a lightweight dynamic analysis on all epochs and outputs the dynamic dependencies that cross epoch boundaries. These inter-epoch dependencies are used to identify the epoch e' by the Comprehensive Analyzer, which then analyzes e' and allows the developer to continue inspecting the recorded execution.

For TESSERACT to be applicable in real-world debugging scenarios, I developed the technique to achieve the following design goals. First, I designed TESSERACT to reduce developers' waiting time before starting the interactive debugging session. To do that, I use a low-overhead record-replay system as the Recorder component. In comparison to systems that record whole traces of executions, TESSERACT substantially reduces the waiting time before the live debugging sessions start. Moreover, the Inter-epoch Analyzer needs to process a potentially large number of epochs and aggregate the results from all epochs. The analysis must complete before the developer can inspect any epoch other than the last one. To reduce the waiting time, I designed this analysis to be highly parallelizable and scale out to large computing clusters. Second, TESSERACT should be efficient in terms of resource usages. To achieve this goal, TESSERACT runs the resource-intensive Comprehensive Analyzer in an on-demand fashion, on only the epochs that are actually inspected by the developer. On the other hand, the Inter-epoch Analyzer, which produces analysis data that support the on-demand approach, runs a more lightweight analysis and requires only a fraction of storage space relative to the full DDG. Third, during the interactive debugging session, TESSERACT should have short response time comparable to that of traditional symbolic debuggers. One challenge in meeting this design goal is to reduce the waiting time when the request of finding a dynamic dependency predecessor requires analyzing another epoch. To do this fast, the Partitioner creates epochs that are small and can be analyzed by the Comprehensive Analyzer quickly. To further reduce the response time in this case, the Comprehensive Analyzer also pre-loads epochs that are likely to be required based on the current inspected position.

I implemented a TESSERACT prototype that analyzes x86 binaries. For code that is in the SUT and is compiled with debugging information, TESSERACT builds DDGs that has sufficient information to map to source code so that the developers still debug the program at source-code leve. I believe that my approach is general and can be adapted to other instruction sets with moderate engineering effort.

```
int A, B;                                  void foo() {
int main() {                                 // Epoch split
  A = 40;                          (s0)      B = A + 3;              (s4)
  B = A - 1;                       (s1)      if (B == 42) {          (s5)
  if (B != 0) {                    (s2)        ++B;                  (s6)
    foo();                         (s3)      }
    return A + B;                  (s7)    }
  }
}
```

Figure 7.2: Example execution for TESSERACT.

The rest of this section is organized as follows. Section 7.1.1 shows an illustrative example that I use to demonstrate how TESSERACT works. Section 7.1.2 describes the Recorder and Partitioner. Section 7.1.3 gives the details of the Comprehensive Analyzer. Section 7.1.4 discusses the Inter-epoch Analyzer. Section 7.1.5 presents the libraries and frameworks I used to implement TESSERACT.

7.1.1 Illustrative Example

Figure 7.2 shows the example execution. Although TESSERACT runs on the compiled x86 machine code of the program, I show it as source code for conciseness. The program has two functions, main and foo. For ease of reference, the program uses only two global variables A and B. I denote the storage locations of the variables with their names. The execution starts from calling the function main and executes each statement once. I use the names in parentheses following each statement to identify each executed instruction (statement) instance. The numerical suffixes of these names indicate their execution order. The statement instance s_3 maps to the call instruction in the compiled binary and not the return instruction. For illustration, I split the execution into two epochs — $epoch_0$ containing statement instances s_0 to s_3 and $epoch_1$ containing instances s_4 to s_7.

7.1.2 Recorder and Partitioner

TESSERACT uses Arnold [47] for recording and replaying executions. When the program being recorded executes, Arnold logs the inputs from all non-deterministic sources; and during the replay, it supplies those values back to the program to ensure that the program executes the same sequence of instructions and produces the same states. Arnold has low overhead in terms of both the recording time and the storage space required for the recording data. On average, the performance overhead is below 8% for most workloads, and the storage overhead is only a few gigabytes per day when recording even an entire workstation used for development. A replayed execution can be instrumented and analyzed with Pin [48]. In particular, the Comprehensive Analyzer and a part of the Inter-epoch Analyzer are implemented as Pintools.

Using a record-replay system as the Recorder has several benefits. First, low-overhead recording enables the technique to integrate better with the typical real-world software development workflow, which runs regression tests on each version of the software and notifies developers of failing tests as potential starting points of debugging sessions. As the Recorder can be turned on by default for test executions, it allows the developers to start debugging sessions using the technique right after failures occur. Second, deterministic replaying is especially valuable for debugging failures that involve nondeterminism and can not be consistently reproduced. Developers can record every attempt of reproducing the failure and investigate the first successful attempt. Third, during the live debugging session, TESSERACT uses Arnold to create a real process that replicates the recorded execution. This allows the developers to inspect the execution in the same way as they would do with a traditional symbolic debugger.

The Partitioner splits the recorded execution into epochs, each of which can be analyzed independently by the Comprehensive Analyzer and the Inter-epoch Analyzer. To do that, the Partitioner uses the *selective instrumentation* mechanism provided by Arnold. To apply a dynamic analysis on a single epoch, the Partitioner first replays the recorded exe-

cution without instrumentation from either the start or a previously created checkpoint of the execution, waits until the execution reaches the starting position of the epoch, and then turns on the instrumentation for the analysis. It terminates the replay after the execution reaches the end of the epoch. To better support analyzing the epochs in parallel, the Partitioner tries to create the epochs such that it would take roughly the same amount of time to analyze each of them. As an approximation, it assumes that the time for analyzing an epoch is proportional to the replay time of the epoch without instrumentation. Based on this assumption, it uses the replay timing data provided by Arnold to create epochs that are roughly equal in replay time.

7.1.3 Comprehensive Analyzer

For each epoch, the Comprehensive Analyzer produces analysis data that support tracing dynamic dependencies and rewinding program states efficiently. This section first discusses the way it builds the DDG and then describes how it represents and restores historical program states.

Conceptually, TESSERACT builds the DDG with a method proposed by Agrawal and colleagues [43]. The resulting DDG is a directed graph $\{V, E\}$. V is a set of nodes, each of which represents a dynamic instance of an executed instruction (i.e., a *definition*). $E \in V \times V$ is a set of edges that represents the dynamic data and control dependencies among the instruction instances. The use and def sets of an instruction instance contain the storage locations that are read and written by the instruction respectively, including the memory addresses, the registers, and a subset of the bits of the EFLAGS register that potentially affects the behavior of subsequent instructions. TESSERACT represents each node in V as a *dependency node*, which stores the instruction that created the current node, the pointers to its data dependency predecessor nodes, and its immediate control dependency node.

To compute the dynamic data dependencies, TESSERACT keeps track of the most recent definition of every storage location by associating the corresponding dependency node with

the location. It does this by using *shadow memory*. For memory addresses in the def set of a definition, TESSERACT stores the pointer to the dependency node in a two-level page table, which enables the technique to allocate shadow memory spaces for only the memory pages used by the program. For registers, the technique maintains one array of dependency node pointers for each thread. TESSERACT stores the definitions for both the memory and registers at a byte-level granularity, but handles those for each bit of the EFLAGS register separately. Specifically, the technique stores dependency information for the carry flag (CF), the parity flag (PF), the adjust flag (AF), the zero flag (ZF), the sign flag (SF), the direction flag (DF), and the overflow flag (OF). Similar to the way TESSERACT associates dependency nodes with registers, it maintains, in the shadow memory, per-thread arrays for these EFLAGS bits.

For each thread, TESSERACT computes dynamic control dependencies by keeping a stack of control-flow conditions. The top of a stack always points to the dependency node p that represents the current control-flow condition. Any newly created dependency node has p as its control dependency predecessor. I first discuss how the technique maintains the control dependency stack inside a single function call and then describe how it handles multiple calls.

Inside a function call, TESSERACT modifies the control dependency stack after every branching instruction and before every instruction at control-flow merge points are executed. For each branching instruction (e.g., JNE, LOOP) that creates a dependency node p_0, the technique pushes a pointer to p_0 onto the control dependency stack. As a result, the control dependencies of the subsequently executed instructions become p_0. For each instruction i at a control-flow merge point, the technique repeatedly pops the control dependency stack if the top of the stack is created by an branching instruction that is post-dominated by i until this condition is no longer met or the stack contains no conditions that are created in the current function call. TESSERACT computes the post-domination relations by statically analyzing the program before the debugging session starts. The

static analysis is intra-procedural. It takes the program and the required shared libraries as input and outputs, for each function, a map from every instruction to its (static) control dependency instruction and a map from every branching instruction to its post-dominator instruction.

TESSERACT handles control dependencies across function calls by storing the control-flow conditions for each function call in separate frames, which is analogous to call frames in call stacks. Pushing and popping conditions can happen only in the top-most frame. The technique pushes a new frame onto the control dependency stack when a function call starts. The new frame initially contains the control dependency of the call instruction instance since it is the control dependency of all top-level code in the function. After the call returns, TESSERACT pops the frame together with all the control-flow conditions in it.

One challenge in identifying function calls for x86 binaries is that the call (resp., return) instructions are not always used for starting (resp., ending) function calls. Although these special usages of call and return instructions (e.g., getting the value of the instruction pointer) are rare, they are difficult to precisely identify. Moreover, incorrectly pushing or popping frames might corrupt the control dependency stack and invalidate all subsequent dependency computations. To address this challenge, I designed TESSERACT to be resilient to occasional false-positive function calls and returns. Intuitively, my approach is to maintain the frames of the control dependency stack in sync with the frames of the actual call stack. TESSERACT creates a control dependency frame for every function call instruction and associates with it an integer identifier that is equal to the memory address at which the return address of the function call is saved. For conciseness, I call it the *return address save (RAS) location* in the rest of the chapter. Specifically, this is the value of the ESP register right after the call instruction is executed. Before executing a return instruction, which reads the RAS location from the ESP register, TESSERACT uses the ESP value l_{return} to find and pop the control dependency frames whose corresponding call frames would be popped. To do that, the technique compares the integer identifier l_{frame} of the top-most

metadata	*control-flow conditions*
$frame_1$: foo (L_{foo})	s_5 (B == 42) s_2 (B != 0)
$frame_0$: main (L_{main})	s_2 (B != 0)

Figure 7.3: Control dependency stack of example for TESSERACT.

frame of the control dependency stack, pops the frame if $l_{return} \geq l_{frame}$, and repeats these steps until the condition is no longer met. This approach guarantees that the frames that should still be on the stack are never incorrectly popped. For a function that uses call and return instructions for purposes other than function calls, TESSERACT might create extra frames, but such inaccuracies are corrected once the function returns.

Figure 7.3 shows the control dependency stack for the full example execution (i.e., not split into epochs) right before the statement instance s_6 executes. The first column shows the metadata associated with the frames and the second column shows the actual control-flow conditions in each frame. The stack grows from the bottom of the table (higher memory address) to the top (lower address). I denote the RAS location of the call to the function main as L_{main} and that of the call to the function foo as L_{foo}, where $L_{foo} < L_{main}$. When the function foo returns, the return instruction instance reads the return address from a memory location L_r, which must be equal to L_{foo}. Therefore, TESSERACT pops $frame_1$ but leaves $frame_0$ on the stack because $L_{main} > L_r$.

The Comprehensive Analyzer handles dynamic dependencies into previous epochs by creating placeholder DDG nodes for definitions that are not defined in the current epoch. It uses the inter-epoch definitions produced by the Inter-epoch Analyzer to find the actual defining epochs for the placeholder nodes. I will explain how TESSERACT does this in Section 7.1.4. To reduce response time in cases when tracing a dynamic dependency edge requires loading another epoch, the Comprehensive Analyzer pre-loads the epochs that are likely needed in the next n tracing steps. To do that, the technique traverses the DDG from the current node being inspected in the backward direction, looks for placeholder DDG

Storage location A	*Storage location B*
$A.entry_0 : \langle t_0, 40, n_0\rangle$	$B.entry_0 : \langle t_1, 39, n_1\rangle$ $B.entry_1 : \langle t_4, 42, n_4\rangle$ $B.entry_2 : \langle t_6, 43, n_6\rangle$

Figure 7.4: Historical states of example execution for TESSERACT.

nodes within a distance of n, and pre-loads the defining epochs of the placeholder nodes it found.

Although having a DDG allows developers to explore the recorded execution by following the dependency relations between executed instructions, I believe that they need to inspect the concrete program state at the moment when an instruction was executing to better understand its behavior. To support that, the Comprehensive Analyzer stores historical values of the application memory, the registers of each thread, and the shadow memory in a format that allows the technique to efficiently rewinding the program state to any point in time in the current epoch. Specifically, for each storage location m, the technique maintains a list H of tuples $\langle t_i, v_i, n_i\rangle$ representing the fact that the storage location is assigned the concrete value v_i and a dependency node n_i at the time t_i, where the timestamp t_i is a counter that TESSERACT increments for each executed instruction. The list H is sorted by t_i to enable efficient look ups by timestamp. To restore the storage location to an arbitrary time t, the technique uses a binary search algorithm on H to locate the tuple $\langle t_x, v_x, n_x\rangle$ that has the largest timestamp smaller than t. The value v_x is the concrete value of the storage location and the value d_x is the pointer to its dependency node at time t. The time complexity of rewinding the state of a storage location is only $O(logL)$ where L is the size of the historical states list. For all executions in the benchmark, restoring program states incurs no delay that the developer can notice during an interactive debugging session.

Figure 7.4 shows the historical states at the end of the execution in the example. The two columns shows the history entries of the global variables A and B respectively for the full execution (i.e., not split into epochs). The notation t_i represents the timestamp at

which the statement instance s_i was executed, and n_i represents the dependency node it creates. As an example, I show how TESSERACT rewinds the program states to before s_5 is executed. For both locations, the technique looks for the entries that have the largest timestamps smaller than t_5 through binary search and finds $A.entry_0$ and $B.entry_1$. Thus, the values of A and B immediately before s_5 executes are 40 and 42, and the associated dependency nodes are n_0 and n_4, respectively.

Since each storage location can be restored separately, the technique can further reduce the cost of rewinding program states by restoring program states partially and on-demand. Initially, it restores only a small number of storage locations that the technique presents to the developer, and can continue restoring more locations as the developer inspects more parts of the memory space.

The major advantage of restoring the dependency nodes of all storage locations is that TESSERACT provides more flexibility to the developers for exploring the recorded executions. After navigating to an instruction instance i on the DDG and rewinding the execution to the time point t when i is executed, the developer not only has the ability of going to one of the dynamic dependency predecessors of i, but also has the choice of tracing the the dependencies from the dependency node of any storage location at t.

The main cost of storing historical program states is the extra storage space, which is comparable to the size of the DDG. I believe that this cost is justified by the benefit of providing near-zero response time during the debugging session. It is also mitigated by the fact that each epoch is small. As I will show in the empirical evaluation, the memory usage of the Comprehensive Analyzer can easily fit on one single personal computer of typical specifications.

7.1.4 Inter-epoch Analyzer

The Inter-epoch Analyzer runs on all epochs of an execution to compute the dynamic data and control dependencies that cross epoch boundaries. The analysis is performed in two

phases — a local analysis phase that processes each epoch individually and an aggregation phase that combines the results from all epochs. Both phases are designed to be highly parallelizable and can run on a computing cluster. I first describe how TESSERACT represents these inter-epoch dependencies, followed by the approach for the local analysis phase and the aggregation phase, and then I discuss how the inter-epoch dependencies are used by the Comprehensive Analyzer to support finding required epochs.

TESSERACT outputs, for each epoch, the definitions that are defined in previous epochs and used in the current epoch in a format that enables the Comprehensive Analyzer to efficiently connect them with downstream DDG nodes. Each of these inter-epoch definitions is annotated with the serial ID of the epoch that defines it. To represent the inter-epoch data dependencies, for each epoch e, the technique outputs a map from the storage locations associated with the inter-epoch definitions used in e to the serial IDs of the defining epochs of these definitions. To represent the inter-epoch control dependencies, TESSERACT outputs, for each thread, a stack of *inter-epoch frames* corresponding to the calls that are started in previous epochs and are *relevant* in the current epoch. A function call is considered to be relevant in an epoch if it has ever become an active call (i.e., on the top of the call stack) at some point in time in the epoch. Each inter-epoch frame stores a map from the branching instructions executed in previous epochs in that frame to the corresponding epoch IDs.

To support computing inter-epoch data dependencies, the local analysis phase produces two types of information: 1) a set U of the storage locations that are read before written in the current epoch and 2) a set D of the storage locations that are defined in the epoch. For convenience in the aggregation phase, D is represented as a map from these storage locations to the epoch ID, which is constant for a given epoch. I designed the local analysis to compute these sets for the convenience of merging these results of neighboring epochs later. Intuitively, the set U contains the locations that reply on and require information from previous epochs, the the map D contains information of the locations that are potentially required by subsequent epochs.

To support computing inter-epoch control dependencies, the local analysis on an epoch produces per-thread stacks of inter-epoch frames for function calls that are relevant in the epoch and potentially still have not returned at the end of the epoch. Because a function call may be relevant in multiple epochs, TESSERACT associates integer identifiers with inter-epoch frames to match them between neighboring epochs. The technique uses values from different sources as identifiers for three different types of frames. First, if a function call is started in the current epoch, TESSERACT uses its RAS location. Second, TESSERACT always creates a frame corresponding to the active call at the moment when the current epoch starts. Its identifier is set to 0. Third, if a function call is started in a previous epoch, and becomes the active call again in the epoch after executing a return instruction instance r, the technique identifies the frame by using the RAS location read by r. I call the frames created for the first case *full inter-epoch frames* and those created for the latter two cases *partial inter-epoch frames*. Each inter-epoch frame contains a map from branching instructions that are executed in that frame in the current epoch to the current epoch ID, which is the same for all entries.

Before describing the aggregation phase, I first explain how TESSERACT merges the local analysis results of two neighboring epochs, which is a basic building block for the aggregation algorithm. Merging the results regarding the storage locations is straightforward. Given two neighboring epochs e_1 and e_2, I denote the U set and the D map computed for each epoch as U_1, U_2, D_1, and D_2. The Inter-epoch Analyzer computes the set U_m and the map D_m for the merged execution range as follows. The set U_m contains all the storage locations in U_1 and the elements in U_2 but not in D_1. The elements of U_2 that are also in D_1 are not added because they are no longer defined by unknown previous epochs as their definitions are already found in the epoch e_1. The map D_m is a union of D_1 and D_2. For keys that appear in both maps, only the entry from D_2 is kept as the map stores the latest definitions of storage locations.

TESSERACT merges the corresponding stacks of inter-epoch frames S_1 and S_2 from e_1

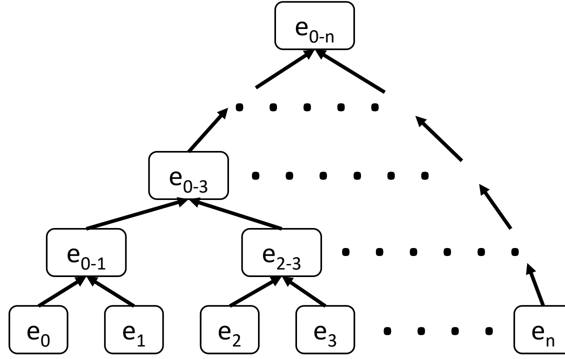

Figure 7.5: Example of tree of epoch ranges for parallel scan.

and e_2 by simulating the pushes and pops of frames. The merged stack S_m of inter-epoch frames starts with the same state as S_1. The technique first merges the partial frames in S_2. For a partial frame f with an identifier l, TESSERACT modifies S_m as if a return instruction that uses an RAS location l is executed and then adds all entries in the frame to the top-most frame of S_m. After merging all partial frames, the technique merges the full frames in S_2. Because these frames only exists in e_2, it simply pushes each full frame onto S_m.

In the aggregation phase, the Inter-epoch Analyzer merges the local analysis results of all epochs by using a parallel scan algorithm [49]. For completeness, I briefly describe the computation workflow. Intuitively, the algorithm decomposes the computational tasks by using a binary tree structure. For n epochs, Figure 7.5 shows an illustrative example of the structure, in which the leaf nodes represent individual epochs and the internal nodes represent the epoch ranges created by merging two neighboring ranges of the lower level. The notation $e_{x\text{-}y}$ represents an execution range from epoch e_x to epoch e_y inclusively. The parallel scan algorithm traverses the tree in a bottom-up pass followed by a top-down pass. The bottom-up pass computes merged local analysis results for each execution range in the tree by recursively merging the results associated with its children. Then the top-down pass uses the intermediate results computed in the previous pass to compute the merged results of each prefix of the epoch sequence $\{e_{0\text{-}i} \mid i = 0, 1, 2, 3, 4, 5, ..., n-1\}$. With sufficient parallelism, the running time of the algorithm is O(log n) (i.e., proportional to the height of

$epoch_0$	$epoch_1$	*Merged*
$U_0 : \emptyset$ $D_0 : \{A : 0, B : 0\}$	$U_1 : \{A\}$ $D_1 : \{B : 1\}$	$U_m : \emptyset$ $D_m : \{A : 0, B : 1\}$

Figure 7.6: Merging local analysis results of storage locations.

$epoch_0$	$epoch_1$	*Merged*
$F_1 : L_{foo} : \emptyset$ $F_0 : L_{main} : \{I(s_2) : 0\}$ $P_0 : 0 : \emptyset$	$P_1 : 0 : \emptyset$ $P_2 : L_{foo} : \emptyset$	$F_0' : L_{main} : \{I(s_2 : 0)\}$ $P_0 : 0 : \emptyset$

Figure 7.7: Merging inter-epoch frames.

the tree).

TESSERACT uses the merged local analysis result of a prefix epoch range $e_{0\text{-}i}$ to compute the inter-epoch dependencies of the epoch e_{i+1}. The inter-epoch data dependencies of e_{i+1} include the entries of the map D of the prefix range filtered by the storage locations in the set U of e_{i+1}. The inter-epoch control dependency data of e_{i+1} are represented with the subset of the inter-epoch frames of the prefix range that match the partial inter-epoch frames of e_{i+1}, as those are the frames that correspond to the function calls that are created in the prefix range and are relevant in the epoch e_{i+1}. Because the prefix execution range start from the first epoch e_0, the resulting list of inter-epoch frames contains only full frames.

I show the local analysis results for both epochs in the example execution, how TESSERACT merges them, and how the technique computes the inter-epoch dependencies for $epoch_1$. To clarify, for a real execution of only two epochs, the technique does not need to merge the local analysis results as the execution ends at the second epoch. I describe it for the example only to illustrate the process. Figure 7.6 shows the U and D sets of each epoch in the example and the merged results. The storage location A is not added to U_m because it also appears in D_0. The merged map D_m indicates that the most recent definition of A is in $epoch_0$ and, on the other hand, the most recent definition of B is in $epoch_1$. Figure 7.7 shows the inter-epoch frames of both epochs and the merged frames. I repre-

sent each frame with 3 fields separated by colons — the name of the frame, the identifier, and the map of the executed branching instructions. The capital letter "F" (resp., "P") in a frame name indicates that the frame is a full (resp., partial) inter-epoch frame. The notation $I(s_2)$ represents the (static) statement corresponding to the executed instance s_2. The first epoch has one partial frame and two full frames. The partial frame P_0 represents the caller of the main function and is not used in the example. The full frames F_0 and F_1 correspond to the function calls of main and foo, respectively. The map entry in F_0 indicates that the branching statement $I(s_2)$ is executed in $epoch_0$. The second epoch contains two partial frames. P_1 corresponds to the function call that $epoch_1$ started in, and P_2 is created when the program returns from the function foo. The local analysis on $epoch_1$ does not have the complete information about these function calls. In particular, because TESSERACT do not assume that return instructions are always used for returning from function calls, it does not pop P_1 from the stack when creating P_2. To compute the merged inter-epoch frames, TESSERACT starts with the frames in $epoch_0$ and merges each frame in $epoch_1$. To merge P_1, the technique simply adds all entries of the frame to the top-most frame in $epoch_0$. And to merge P_2, TESSERACT first modifies the stack in $epoch_0$ as if a return instruction with the RAS location of L_{foo} is executed, resulting in popping F_1, and then adds all entries in P_2 to F_0, which is the top-most frame after the pop operation. TESSERACT computes the final inter-epoch dependencies for $epoch_1$ by filtering the results of $epoch_0$ (the prefix range for $epoch_1$). Since U_1 contains one single storage location A, its inter-epoch data dependencies include one entry $\{A : 0\}$. Its inter-epoch control dependencies include the frames F_0 and F_1 in $epoch_0$ because they matches the partial frames P_2 and P_1, respectively.

The Comprehensive Analyzer uses the inter-epoch dependencies by initializing the states of the shadow memory and the control dependency stacks at the start of the analysis of each epoch. For each data dependency entry $\langle l, ID(e) \rangle$, the Comprehensive Analyzer creates a placeholder dependency node that contains the epoch id $ID(e)$ and associate it with the storage location l. For each inter-epoch frame f, the technique pushes a cor-

responding control dependency frame f' to the stack. During the dynamic analysis, the situation in which f' is the top-most frame and does not contain any control-flow conditions indicates that the control dependency node of the current instruction instance i is created in a previous epoch e'. If i is not top-level code of the current function, TESSERACT looks up the ID of e' in the inter-epoch frame f by using the (static) control dependency instruction of i as the key. Otherwise, if i is top-level code, its control dependency node must be in the caller function. In this case, e' is the epoch that has the largest ID in all the entries of the inter-epoch frame of the caller function. TESSERACT then pushes a new placeholder dependency node that contains the ID of e' to the control dependency stack. The placeholder nodes created for both data and control dependencies are used in the same way as regular dependency nodes to build the DDG. During the interactive debugging session, the Comprehensive Analyzer loads the corresponding epochs when either the developers' dependency-tracing requests or the look-ahead search of the pre-loading feature reach placeholder nodes.

7.1.5 Implementation

I implemented the local dynamic analyses of TESSERACT through dynamic binary instrumentation by using Intel Pin [48] and Intel XED [50]. To compute the static control dependencies, the technique uses the angr binary analysis framework [51, 52, 53]. The technique extracts and processes the debugging information in binaries with the DWARF parser of LLVM [54]. TESSERACT handles serialization for both persistent data storage and network communication by using the Boost serialization library [55]. I implemented the parallel jobs in the aggregation phase of the Inter-epoch Analyzer with Open MPI [56].

7.2 Empirical Evaluation

I evaluated TESSERACT on a benchmark of real-world applications to answer the following research questions:

Table 7.1: Benchmark executions for TESSERACT.

Program	*Length (ms)*	*# Dep Nodes (M)*	*# Dep Edges (M)*
Grep	124	268.8	790.6
Gzip	2460	10690.6	11045.2
Sha512sum	656	3135.6	5548.6
Factor	8	55.2	78.4
Seq	456	1252.5	2049.7
Sed	1052	3426.7	5647.8

RQ1: How well does TESSERACT scale on the benchmark executions?

RQ2: How well does the Comprehensive Analyzer support interactive debugging?

7.2.1 Experiment Setup

Table 7.1 shows a summary of the benchmark executions. The programs Gzip and Sed are from the SIR repository [57] and the other 4 programs are from the Corebench repository [58]. I compiled the benchmark programs as x86 debug-version binaries with GCC 4.8. I did not use the unit tests provided by the repositories because they are too short to show the effectiveness of TESSERACT, but instead manually created one execution for each program. I recorded the executions of running Grep, Gzip, Sha512sum, and Sed on randomly generated large files, running Factor to compute the factors of a large number $(2^{31} - 1) \times (2^{61} - 1)$, and running Seq to print numbers from 1 to $10^7 - 1$. The column "Length" shows the replay time of each execution without instrumentation in milliseconds. I included executions of varying lengths from 8 milliseconds to 2460 milliseconds to evaluate the performance of TESSERACT in these different situations. The column "# Dep Nodes" shows the number of DDG nodes of each execution and the column "# Dep Edges" shows the number of DDG edges. The numbers in both these columns are shown in millions. Except for the benchmark executions of the programs Factor and Grep, the DDG sizes of all other executions are from dozens to hundreds of gigabytes, which is beyond what traditional single-node dynamic dependency analysis tools can handle.

I ran the experiment on a CloudLab [59] computing cluster of 64 m510 nodes. Each node has an 8-core Intel Xeon D-1548 2.0 GHz CPU, 64GB ECC memory (4 $\times$ 16GB DDR4-2133 SO-DIMMs), 256 GB NVMe flash storage, and 10 GB NIC. Because of implementation limitations, a part of the Inter-epoch Analyzer runs in virtual machines. On each node, I allocated 4 cores to the virtual machine and at any given time, TESSERACT runs on only 4 cores. This gives the technique 256 cores in total. Although TESSERACT continues to scale with more cores, the cluster of 64 machines is the largest one that I can consistently get from CloudLab.

7.2.2 RQ1: How well does TESSERACT scale on the benchmark executions?

To evaluate the scalability of TESSERACT, I measured the running time of the Inter-epoch Analyzer on exponentially increasing numbers of CPU cores from 1 to 256 (i.e., 1, 2, 4, 8, ..., 256). I ran each experiment for 5 times and used the average running time of all trials. I computed the *speedup ratio* of utilizing x cores by dividing the running time of the Inter-epoch Analyzer on a single core by its running time on x cores.

Figure 7.8 shows the speedup ratios of the Inter-epoch Analyzer on varying number of cores for each benchmark execution. The dotted black line shows the "ideal" case where the performance of the system increases proportionally with the available cores. Both the axes are displayed in log scale since the range of the values are large. I could not run the benchmark execution of the program Factor on more than 32 cores because the execution is short and maximum number of epochs that the Partitioner can create is less than 64. For the execution of the program Grep, because running the analysis on more than one core on a machine crashes the Arnold system, I ran the experiment for this program on up to 64 cores (i.e., one core per node). The result shows that TESSERACT scales well with increased computing resources. Its scalability is positively correlated with the length of the execution being analyzed. For the four longer benchmark executions, with 256 cores, the speedup ratio ranges from 18.5 times to 100.7 times. The average speedup ratio of these

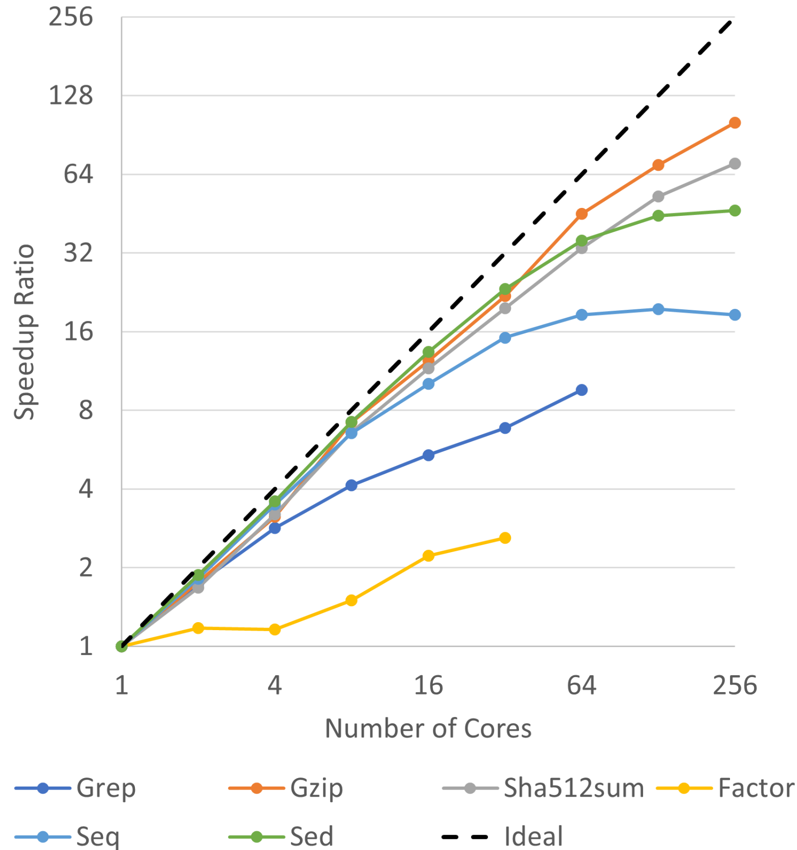

Figure 7.8: Speedup ratio of Inter-epoch Analyzer from 1 to 256 cores.

executions is 59 times.

I show the running time of the Inter-epoch Analyzer in Table 7.2. The column “Sequential” shows the time to run the analyzer with a single core in seconds. The column “Max # Cores” shows the maximum number of cores I have used to run the analyzer on that execution, and the column “Parallel” shows its running time with this number of cores. With the maximum parallelism, the Inter-epoch Analyzer completed for 4 of the 6 executions within 6 seconds. The analysis took longer to finish for the other two executions but, as the data in Figure 7.8 indicates, it continues to speed up with more than 256 cores. This result shows that the Inter-epoch Analyzer provides sufficiently good performance to support a typical debugging workflow. Although the most time-consuming analysis took 14 seconds, the developer will likely still experience no waiting time, as the analysis can run in the background while the developer is inspecting the last epoch.

Table 7.2: Running time of Inter-epoch Analyzer.

Program	*Sequential (s)*	*Max # Cores*	*Parallel (s)*
Grep	34	64	4
Gzip	1390	256	14
Sha512sum	647	256	9
Factor	11	32	4
Seq	97	256	5
Sed	272	256	6

Table 7.3: Storage overhead of Inter-epoch Analyzer.

Program	*Total Size (MB)*			*Count per Epoch*			
	DDG	*Inter Def*	*Inter Def / DDG*	*# Intra Def*	*# Inter Def*	*Inter / Intra*	*Pearson*
Grep	7,116.8	0.98	0.014%	2,079,878	3,829	0.184%	0.20
Gzip	205,259.4	377.65	0.184%	4,037,325	89,888	2.226%	0.10
Sha512sum	69,011.4	0.48	(< 0.001%)	11,176,243	206	0.002%	0.11
Factor	1,140.7	0.02	0.002%	3,177,140	195	0.006%	-0.27
Seq	26,931.0	0.52	0.002%	2,666,118	388	0.015%	0.16
Sed	73,832.5	2.19	0.003%	3,222,839	868	0.027%	0.11
Average	63,882.0	63.6	0.034%	4,393,257	15,896	0.410%	-

Among all benchmark programs, TESSERACT scales the least well for the program Factor. I checked its execution data of individual epochs and identified two reasons. First, the epoch partitioning of this execution is imbalanced because of the limitation of the underlying replay system. The longest epoch dominates the running time and thus reduces the effectiveness of parallelization. I also checked the partitioning of the other subject executions and found that this type of imbalances occur only infrequently. Second, since the entire execution is short, further splitting the workload leads to the situation where the overhead of parallelization consumes most of the computing time. Under the 32-core configuration, the local analysis at each node only took 17% of the total running time while the parallel tasks took 83%. In comparison, for executions that scales well with many cores such as Gzip, Sha512sum, the local analysis accounts for more than 55% of total running time even on a 256-core cluster.

The storage size of the inter-epoch definitions produced by TESSERACT also affects

how well it handles real-world executions. The column group “Total Size” in Table 7.3 shows the storage overhead of the technique. In this group, the column “DDG” shows, as a baseline of comparison, the size of the DDG created by the Comprehensive Analyzer for each benchmark execution, the column “Inter Def” shows the compressed size of the inter-epoch definitions, and the column “Inter Def / DDG” shows the ratio of the inter-epoch definition size over the full DDG size as a percentage. All sizes in the column group are displayed in megabytes. The storage required for the inter-epoch definitions for each execution ranges from 20KB to about 377MB. The average storage consumption is 63.6MB. In contrast, the full DDGs can be as large as several hundred gigabytes, and the average is about 62 gigabytes for all executions in the benchmark. The inter-epoch definitions require only a fraction of the size of the DDGs to store. The size ratios are less than 0.2% for all executions, and the average ratio is only about 0.03%.

To understand the storage usage of the Inter-epoch Analyzer, the column group “Count per Epoch” in Table 7.3 reports the average number of definitions that are created in each epoch and the average number of inter-epoch definitions used in each epoch in the columns “# Intra Def” and “# Inter Def” respectively. The overall average number of definitions created per epoch across all executions is about 4.4 million, while the average number of inter-epoch definitions per epoch is around 16 thousand. The column “Inter / Intra” in this group reports the ratio between the “# Inter Def” and “# Intra Def” columns, which is on average only 0.4%. This result indicates that the number of inter-epoch dependencies in the DDG are far less than the intra-epoch ones, despite having a large number of epochs. My hypothesis is that this is a manifestation of the *principle of locality* [1]. Since this is the phenomenon that also underpins the hierarchical memory design of modern computer architecture, I expect the low ratio between inter- and intra-epoch dependencies to occur in the majority of program executions.

I further investigated whether the ratio of inter- and intra-epoch dependencies correlates

[1] Also known as locality of reference [60]. In a short period of time, the processor tends to use the same set of storage locations repeatedly.

with the relative position of an epoch in an execution. This is a plausible hypothesis because, compared to an epoch near the beginning of an execution, an epoch near the end has a higher chance of referencing a data item defined in previous epochs since more code has been executed before it. Having an unbalanced distribution of inter-epoch dependencies, however, would negatively affect the performance of TESSERACT. To measure the strength of the correlation, for each execution, I assigned a serial number to each of the epochs in temporal order, and computed the Pearson coefficient between the epoch number and its inter-epoch dependency ratio. The column "Pearson" in Table 7.3 reports the result. The absolute values of the Pearson coefficients range from 0.1 to 0.27, indicating no correlation.

7.2.3 RQ2: How well does the Comprehensive Analyzer support interactive debugging?

During the live debugging session, in response to developers' requests of tracing dynamic dependencies, the Comprehensive Analyzer needs to traverse the dependencies and rewinding program states fast. The main metric I used to evaluate this component is its response time to developers' requests. Moreover, because the component trades space for time through its pre-loading feature, I also discuss its memory usage.

I measured its performance on the finest-grained epochs that I created in the experiment for answering RQ1. Except for the execution of Factor, I partitioned all other executions into epochs of, on average, 1 millisecond in replay time without instrumentation. I created smaller epochs for the execution of Factor. Specifically, it is partitioned into 32 epochs so that the Inter-epoch Analyzer can run it on 32 cores. Table 7.4 shows the results. I first reports the performance metrics when responding to the requests does not require analyzing previous epochs and then discuss the performance when it does.

Inside a single epoch, finding dependency predecessors is trivial as TESSERACT has built the DDG. I only report the performance for rewinding program states. I randomly picked 32 timestamps for each epoch of each execution, restored all storage locations to each of these time points, and measured the time spent on each rewinding operation. The

Table 7.4: Response time and memory consumption of Comprehensive Analyzer.

Program	*RT*	*Mem/Epoch*			*AT (s)*	*Percent DDG Nodes*	
		DDG	*History*	*Total*		*Preloads* ≤ 3	*Preloads* > 3
Grep	9.48	55.6	95.3	150.9	5.2	97.563%	2.437%
Gzip	10.23	83.4	161.8	245.3	6.1	>99.999%	<0.001%
Sha512sum	10.48	105.4	508.9	614.2	19.3	>99.999%	<0.001%
Factor	9.53	35.6	36.6	72.2	3.0	>99.999%	<0.001%
Seq	9.70	59.1	115.7	174.8	4.8	99.698%	0.302%
Sed	9.62	70.2	133.7	203.9	5.6	99.960%	0.040%
Average	9.84	68.2	175.3	243.5	7.3	99.537%	0.463%

column "RT" of Table 7.4 reports the average rewinding time in milliseconds for each benchmark execution. The time spent is on average 9.84 ms and consistent for all executions. This result shows that, on the benchmark, rewinding program states is fast and causes no delay that developers can notice during the live debugging session.

I report the sizes of the dynamic analysis data produced by the Comprehensive Analyzer in the column group "Mem/Epoch". The column "DDG" shows the average DDG size of each epoch, the column "History" shows the average size of the historical program states per epoch, and the column "Total" is the average total memory usage of the component per epoch. All three columns show sizes in megabytes. The average memory usage per epoch ranges from 72 MB to 614 MB and the overall average of all executions is 244 MB. This result shows that the Comprehensive Analyzer uses a moderate amount of memory and can run locally on the developer's workstation during the live debugging session.

When following a dynamic dependency edge requires analyzing another epoch, the developer needs to wait for the analysis to finish if the pre-loading feature is turned off. The column "AT" in Table 7.4 shows the average time of analyzing each epoch. Except for the execution of Sha512sum, the analysis finishes within about 6 seconds. For all executions, the average analysis time is 7.3 seconds. The pre-loading feature, which looks ahead in the DDG and loads epochs that are likely needed in advance, reduces developers' waiting time in this case. Although the effectiveness of pre-loading depends on the potentially

complex behavior of the developer during debugging, I provide an estimate by making a few assumptions about developer behavior. I assume that the developer 1) uses the tool by only following dynamic dependencies, and 2) spends 10 seconds at each step to understand the behavior of the statement and identify the next dynamic dependency predecessor to look at. I configured the Comprehensive Analyzer to find the epochs for pre-loading by looking ahead 3 levels in the DDG. Under my assumptions, these are the epochs that are likely needed in the next 30 seconds, which is sufficient for analyzing an epoch in the background while the developer inspects the current epoch. To constrain the memory usage, I set the maximum number of epochs that TESSERACT pre-loads to 3. For each DDG node, I computed the number of epochs to pre-load and check whether it is within the limit of 3 epochs. The column group "Percent DDG Nodes" reports the result. The column "Preloads ≤ 3" shows the percentage of DDG nodes that require pre-loading less than or equals to 3 epochs, and the column "Preloads > 3" shows the percentage of nodes that need to preload more than 3 epochs. On average, TESSERACT needs to pre-load less than or equal to 3 epochs at over 99.5% of the DDG nodes (or execution steps). In these cases, developers' waiting time is still zero. In the other 0.5% cases, the developer may need to wait for the technique to analyze an epoch, but these are the rare cases.

7.2.4 Limitations and Threats to Validity

TESSERACT does not fully track the obsolete X87 floating point number stack (i.e., ST0 to ST7) because of the extra performance cost for implementing it and the fact that these registers are usually not used by new software.

The main internal threat is that my implementation of TESSERACT may contain faults. To mitigate this threat, I carefully tested my code with both unit-level and integration tests throughout the development process. I also manually spot checked the outputs of the tool against my manual estimates.

One external threat is that the benchmark executions might not be representative, caus-

ing my results not to generalize. To mitigate this threat, I used programs that perform a wide range of types of tasks such as text processing, compressing, and hashing. I also included executions of varying lengths to investigate the performance of TESSERACT in these different situations. Another external threat is that the evaluation of the Inter-epoch Analyzer of TESSERACT uses a shared computing cluster, where the performance measurements may be affected by random events that are outside of my control. To mitigate this threat, I ran each experiment for 5 times and used the average of all trials.

CHAPTER 8
CONCLUSION

The goal of my research is to define and improve automated debugging techniques to support a develop-centric debugging process for real-world software development activities. To achieve this goal, I defined three techniques: 1) ENLIGHTEN, which is an interactive feedback-driven automated fault localization approach, 2) the potential memory-address dependences, which improves the accuracy of dynamic slicing to better support software debugging, and 3) TESSERACT, which addresses the scalability limitation of existing dependency-based debugging techniques.

Enlighten supports and automates developers' debugging workflow by 1) using traditional statistical fault localization to formulate an initial hypothesis of the fault, 2) identifying a relevant subset of execution that can help support or refute the formulated hypothesis, 3) presenting the developer with a query about the identified execution subset in the form of a correctness question about the input-output relation of the partial execution, 4) refining its hypothesis of the fault by using developers' feedback, and 5) repeating these steps until the fault is found. Enlighten overcomes the important limitation of traditional automated debugging techniques that output only a list of suspicious code entities in that it does not require developers to determine the correctness of statements in isolation, but rather to check high-level input-output relations in concrete executions. Enlighten gets developers' input at a level of abstraction they can typically understand and, when successful, nicely guides the developers towards the fault by following an iterative process.

The second technique I presented introduces the potential memory-address dependence (PMD) and describes an algorithm to compute it. The technique addresses the limitation of existing dynamic dependence analysis approaches defined by dynamic and relevant slicing techniques, that they might miss the dependence paths from faults to manifestations if in-

correct program states are caused by faulty memory addresses. The technique encodes the causal relation as a PMD relation between a variable v used to compute memory addresses and the memory locations that (1) are not assigned but (2) would be assigned on the same control flow path if v had taken different values. Considering PMD enables existing automated or interactive debugging techniques that rely on dynamic dependence analysis to work on a broader range of faults.

TESSERACT makes existing dependency-based debugging techniques scale to real-world executions. By leveraging recent advancements in record-replay systems, the technique speeds up the traditionally sequential dynamic dependency analysis by using massive parallelism and computing clusters. To support debugging a failing execution, TESSERACT uses a record-replay system to efficiently reproduce the execution, splits the execution into small epochs that can be independently analyzed, runs a light-weight dynamic analysis on all epochs in parallel on the computing cluster, and applies the expensive part of the analysis only on the epochs that are actually needed during the debugging process.

To further improve my techniques and better support the developer-centric automated debugging process, I envision three main directions for future work. One research direction is to improve the user-interaction method for debugging. One way to do that is to suitably integrate the two interaction methods involved in my techniques, each of which has its advantages. Similar to ENLIGHTEN, such a technique should be able to jump across an execution to show developers only the most suspicious parts while provide rich execution context (e.g., method call inputs and outputs) for developers to understand the selected parts. On the other hand, such a technique should also provide the dependency-based interaction method, which is assumed by the PMD analysis and TESSERACT, to help developers follow dynamic program dependencies to reason about the behavior of the selected parts of the execution at the finest granularity. The second research direction is to improve the computational efficiency of my techniques, which all involve expensive dynamic analyses. One potential approach is to utilize static program analysis techniques to reduce the amount

of dynamic information that must be computed by filtering the execution to exclude certain parts or approximating the semantics of library functions, which can be assumed to be correct, with less expensive computations. The third direction is to investigate how automated program repair can be improved in the developer-centric debugging process. Such a technique could utilize the unique advantages in this interactive setting, including immediate feedback from developers about a proposed repair candidate and the ability of gathering more information about the program specification interactively.

REFERENCES

[1] J. W. Lloyd, “Declarative error diagnosis”, *New Generation Computing*, pp. 133–154, 1987.

[2] E. Y. Shapiro, *Algorithmic Program DeBugging*. Cambridge, MA, USA: MIT Press, 1983.

[3] J. A. Jones, M. J. Harrold, and J. Stasko, “Visualization of test information to assist fault localization”, in *Proceedings of the 24th International Conference on Software Engineering*, ser. ICSE 2002, 2002, pp. 467–477.

[4] L. Gong, D. Lo, L. Jiang, and H. Zhang, “Interactive fault localization leveraging simple user feedback”, in *Proceedings of the 28th International Conference on Software Maintenance*, ser. ICSM 2012, 2012, pp. 67–76.

[5] A. J. Ko and B. A. Myers, “Designing the whyline: A debugging interface for asking questions about program behavior”, in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI 2004, 2004, pp. 151–158.

[6] Y. Lin, J. Sun, Y. Xue, Y. Liu, and J. Dong, “Feedback-based debugging”, in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE 2017, 2017, pp. 393–403.

[7] Z. Xu, S. Ma, X. Zhang, S. Zhu, and B. Xu, “Debugging with intelligence via probabilistic inference”, in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE 2018, 2018, pp. 1171–1181.

[8] D. Hao, L. Zhang, T. Xie, H. Mei, and J.-S. Sun, “Interactive fault localization using test information”, *Journal of Computer Science and Technology*, vol. 24, no. 5, pp. 962–974, 2009.

[9] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, “Scalable statistical bug isolation”, in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2005, 2005, pp. 15–26.

[10] X. Zhang, N. Gupta, and R. Gupta, “Locating faults through automated predicate switching”, in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE 2006, 2006, pp. 272–281.

[11] W. Eric Wong, V. Debroy, and B. Choi, “A family of code coverage-based heuristics for effective fault localization”, *Journal of Systems and Software*, vol. 83, pp. 188–208, 2010.

[12] C. Parnin and A. Orso, “Are automated debugging techniques actually helping programmers?”, in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA 2011, 2011, pp. 199–209.

[13] Q. Wang, C. Parnin, and A. Orso, “Evaluating the usefulness of ir-based fault localization techniques”, in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015, Baltimore, MD, USA, 2015, pp. 1–11.

[14] Y. Lei, X. Mao, X. Wan, and C. Wang, “Iterative feedback-based fault localization approach”, in *2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications*, ser. SEAA 2011, 2011, pp. 349–356.

[15] H. Cleve and A. Zeller, “Locating causes of program failures”, in *Proceedings of the 27th International Conference on Software Engineering*, ser. ICSE 2005, 2005, pp. 342–351.

[16] M. Beller, N. Spruit, D. Spinellis, and A. Zaidman, “On the dichotomy of debugging behavior among programmers”, ser. ICSE ’18, Gothenburg, Sweden, 2018, pp. 572–583.

[17] X. Zhang, H. He, N. Gupta, and R. Gupta, “Experimental evaluation of using dynamic slices for fault location”, in *Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging*, ser. AADEBUG 2005, 2005, pp. 33–42.

[18] A. J. Ko and B. A. Myers, “Finding causes of program output with the java whyline”, in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI 2009, 2009, pp. 1569–1578.

[19] T. Gyimóthy, Á. Beszédes, and I. Forgács, “An efficient relevant slicing method for debugging”, in *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE 1999, 1999, pp. 303–321.

[20] X. Zhang, S. Tallam, N. Gupta, and R. Gupta, “Towards locating execution omission errors”, in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2007, 2007, pp. 415–424.

[21] A. J. Ko and B. A. Myers, “Debugging reinvented: Asking and answering why and why not questions about program behavior”, in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE 2008, 2008, pp. 301–310.

[22] A. Quinn, D. Devecsery, P. M. Chen, and J. Flinn, “Jetstream: Cluster-scale parallelization of information flow queries”, in *12th USENIX Symposium on Operating*

Systems Design and Implementation (OSDI 16), ser. USENIX 2016, 2016, pp. 451–466.

[23] *Enlighten tool and experiment infrastructure*, http://www.cc.gatech.edu/~orso/software/enlighten/.

[24] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. C. van Gemund, “A practical evaluation of spectrum-based fault localization”, *Journal of Systems and Software*, vol. 82, pp. 1780–1792, 2009.

[25] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, “Evaluating and improving fault localization”, in *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, 2017, pp. 609–620.

[26] B. Korel and J. Laski, “Dynamic program slicing”, *Information Processing Letters*, vol. 29, pp. 155–163, 1988.

[27] H. Agrawal and J. R. Horgan, “Dynamic program slicing”, in *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, ser. PLDI 1990, 1990, pp. 246–256.

[28] X. Li, S. Zhu, M. d’Amorim, and A. Orso, “Enlightened debugging”, in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE 2018, 2018, pp. 82–92.

[29] J. A. Jones, J. F. Bowring, and M. J. Harrold, “Debugging in parallel”, in *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ser. ISSTA 2007, London, United Kingdom, 2007, pp. 16–26.

[30] B. Ashok, J. Joy, H. Liang, S. K. Rajamani, G. Srinivasa, and V. Vangala, “Debugadvisor: A recommender system for debugging”, in *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC/FSE 2009, Amsterdam, The Netherlands, 2009, pp. 373–382.

[31] R. Abreu, P. Zoeteweij, and A. J. C. v. Gemund, “An evaluation of similarity coefficients for software fault localization”, in *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing*, ser. PRDC 2006, 2006, pp. 39–46.

[32] P. D. Stotts, M. Lindsey, and A. Antley, “An informal formal method for systematic junit test case generation”, in *Proceedings of the Second XP Universe and First Agile Universe Conference on Extreme Programming and Agile Methods*, ser. XP/Agile Universe 2002, 2002, pp. 131–143.

[33] *SAEG - Software Analysis and Experimentation Group (at Universidade de São Paulo (USP), Brazil)*, https://github.com/saeg/experiments/tree/master/jaguar-2015, 2015.

[34] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs", in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014, 2014, pp. 437–440.

[35] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model checking programs", *Automated Software Engineering*, vol. 10, pp. 203–232, 2003.

[36] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?", in *Proceedings of the 27th International Conference on Software Engineering*, ser. ICSE 2005, St. Louis, MO, USA, 2005, pp. 402–411.

[37] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?", in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014, Hong Kong, China, 2014, pp. 654–665.

[38] R. Just, "The Major mutation framework: Efficient and scalable mutation analysis for Java", in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014, Jul. 2014, pp. 433–436.

[39] *Java instrumentation api (class redefinition)*, https://docs.oracle.com/javase/8/docs/api/java/lang/instrument/package-summary.html, 2017.

[40] K. Pearson, "Notes on regression and inheritance in case of two parents", in *Royal Society of London*, Jun. 1895, pp. 246–263.

[41] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?", in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA 2011, Toronto, Ontario, Canada, 2011, pp. 199–209.

[42] X. Li and A. Orso, "More accurate dynamic slicing for better supporting software debugging", in *Proceedings of the 13th International Conference on Software Testing, Validation and Verification (ICST)*, ser. ICST 2020, 2020, pp. 28–38.

[43] H. Agrawal, R. A. DeMillo, and E. H. Spafford, "Dynamic slicing in the presence of unconstrained pointers", in *Proceedings of the Symposium on Testing, Analysis, and Verification*, ser. TAV 1991, 1991, pp. 60–73.

[44] *Java reflection api*, https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/package-summary.html.

[45] W. Visser, C. S. Păsăreanu, and S. Khurshid, "Test input generation with java pathfinder", in *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2004, 2004, pp. 97–107.

[46] *Identity hash code*, https://docs.oracle.com/javase/8/docs/api/java/lang/System.html.

[47] D. Devecsery, M. Chow, X. Dou, J. Flinn, and P. M. Chen, "Eidetic systems", in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI 2014, 2014, pp. 525–540.

[48] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation", in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2005, 2005, pp. 190–200.

[49] R. E. Ladner and M. J. Fischer, "Parallel prefix computation", *Journal of the ACM*, vol. 27, no. 4, pp. 831–838, 1980.

[50] *Intel x86 encoder decoder (intel xed)*, https://github.com/intelxed/xed, 2021.

[51] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "Sok: (state of) the art of war: Offensive techniques in binary analysis", 2016.

[52] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution", 2016.

[53] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware", 2015.

[54] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation", in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, ser. CGO 2004, 2004, p. 75.

[55] *Boost serialization library*, https://www.boost.org/doc/libs/1_75_0/libs/serialization/doc/index.html, 2021.

[56] R. L. Graham, T. S. Woodall, and J. M. Squyres, "Open mpi: A flexible high performance mpi", in *Proceedings of the 6th International Conference on Parallel Processing and Applied Mathematics*, ser. PPAM 2005, Poznań, Poland, 2005, pp. 228–239.

[57] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact", *Empirical Software Engineering*, vol. 10, pp. 405–435, 2005.

[58] M. Böhme and A. Roychoudhury, "Corebench: Studying complexity of regression errors", in *Proceedings of the 23rd ACM/SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2014, 2014, pp. 105–115.

[59] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra, "The design and operation of CloudLab", in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2019, pp. 1–14.

[60] P. Denning, "The locality principle", in *Communications of the ACM*. 2006, pp. 43–67.

[61] E. Av-Ron, *Top-down Diagnosis of Prolog Programs*. 1984.

[62] A. Tessier and G. Ferrand, "Declarative diagnosis in the clp scheme", in *Analysis and Visualization Tools for Constraint Programming, Constrain Debugging (DiSCiPl Project)*, 2000, pp. 151–174.

[63] L. Naish, "Declarative debugging of lazy functional programs", Australian Computer Science Communications, Tech. Rep., 1992.

[64] O. Shmueli and S. Tsur, "Logical diagnosis ofldl programs", *New Generation Computing*, p. 277, 1991.

[65] L. Naish, "Declarative diagnosis of missing answers", *New Generation Computing*, pp. 255–285, 1992.

[66] L. Naish, "A three-valued declarative debugging scheme", in *Proceedings 23rd Australasian Computer Science Conference.*, 2000, pp. 166–173.

[67] M. P. J. Fromherz, "Towards declarative debugging of concurrent constraint programs", in *Automated and Algorithmic Debugging*, 1993, pp. 88–100.

[68] W. Drabent, S. Nadjm-Tehrani, and J. Maluszynski, "The use of assertions in algorithmic debugging", in *FGCS*, 1988.

[69] R. Caballero, E. Martin-Martin, A. Riesco, and S. Tamarit, “A zoom-declarative debugger for sequential erlang programs”, *Science of Computer Programming*, pp. 104–118, 2015.

[70] F. Russo and M. Sancassani, “A declarative debugging environment for datalog”, in *Logic Programming*, 1992, pp. 433–441.

[71] R. Kowalski and D. Kuehner, “Linear resolution with selection function”, *Artificial Intelligence*, vol. 2, pp. 227–260, 1971.

[72] T. Arora, R. Ramakrishnan, W. G. Roth, P. Seshadri, and D. Srivastava, “Explaining program execution in deductive systems”, in *Deductive and Object-Oriented Databases*, S. Ceri, K. Tanaka, and S. Tsur, Eds., 1993, pp. 101–119.

[73] C. A. Wieland, “Two explanation facilities for the deductive database management system dedex”, in *Proceedings of the 9th International Conference on Entity-Relationship Approach*, 1990.

[74] J. Silva and O. Chitil, “Combining algorithmic debugging and program slicing”, in *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, ser. PPDP 2006, 2006, pp. 157–166.

[75] M. Faddegon and O. Chitil, “Algorithmic debugging of real-world haskell programs: Deriving dependencies from the cost centre stack”, in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2015, 2015, pp. 33–42.

[76] ——, “Lightweight computation tree tracing for lazy functional languages”, in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2016, 2016, pp. 114–128.

[77] O. Chitil and T. Davie, “Comprehending finite maps for algorithmic debugging of higher-order functional programs”, in *Proceedings of the 10th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, ser. PPDP 2008, 2008, pp. 205–216.

[78] P. Fritzson, N. Shahmehri, M. Kamkar, and T. Gyimothy, “Generalized algorithmic debugging and testing”, *ACM Letters on Programming Languages and Systems*, vol. 1, pp. 303–322, 1992.

[79] H.-J. Kouh and W.-H. Yoo, “The efficient debugging system for locating logical errors in java programs”, in *Proceedings of the 2003 International Conference on Computational Science and Its Applications*, ser. ICCSA 2003, 2003, pp. 684–693.

[80] C. Hermanns and H. Kuchen, “Hybrid debugging of java programs”, in *Software and Data Technologies*, 2013, pp. 91–107.

[81] D. Insa and J. Silva, “An algorithmic debugger for java”, in *2010 IEEE International Conference on Software Maintenance*, 2010, pp. 1–6.

[82] J. Silva, “A survey on algorithmic debugging strategies”, *Advances in Enginnering Software*, vol. 42, no. 11, pp. 976–991, Nov. 2011.

[83] R. Caballero, A. Riesco, and J. Silva, “A survey of algorithmic debugging”, *ACM Computing Surveys*, 60:1–60:35, 2017.

[84] X. Li, M. d’Amorim, and A. Orso, “Iterative user-driven fault localization”, in *Proceedings of the 12th International Haifa Verification Conference*, ser. HVC 2016, 2016, pp. 82–98.

[85] M. A. Francel, *Fault localization through execution traces*, 2006.

[86] M. Renieris and S. P. Reiss, “Fault localization with nearest neighbor queries”, in *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, ser. ASE 2003, 2003, pp. 30–39.

[87] T. Ball, M. Naik, and S. K. Rajamani, “From symptom to cause: Localizing errors in counterexample traces”, in *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL 2003, 2003, pp. 97–105.

[88] B. Demsky, M. D. Ernst, P. J. Guo, S. McCamant, J. H. Perkins, and M. Rinard, “Inference and enforcement of data structure consistency specifications”, in *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ser. ISSTA 2006, 2006, pp. 233–244.

[89] A. X. Zheng, M. I. Jordan, B. Liblit, M. Naik, and A. Aiken, “Statistical debugging: Simultaneous identification of multiple bugs”, in *Proceedings of the 23rd International Conference on Machine Learning*, ser. ICML 2006, 2006, pp. 1105–1112.

[90] S. Chandra, E. Torlak, S. Barman, and R. Bodik, “Angelic debugging”, in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE 2011, 2011, pp. 121–130.

[91] M. Jose and R. Majumdar, “Cause clue clauses: Error localization using maximum satisfiability”, in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2011, 2011, pp. 437–446.

[92] J. A. Jones, J. F. Bowring, and M. J. Harrold, "Debugging in parallel", in *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ser. ISSTA 2007, 2007, pp. 16–26.

[93] W. Masri, "Fault localization based on information flow coverage", *Software Testing, Verification and Reliability*, vol. 20, pp. 121–147, 2010.

[94] X. Wang, Q. Gu, X. Zhang, X. Chen, and D. Chen, "Fault localization based on multi-level similarity of execution traces", in *Proceedings of the 2009 16th Asia-Pacific Software Engineering Conference*, 2009, pp. 399–405.

[95] Y. Chung, C. Huang, and Y. Huang, "A study of modified testing-based fault localization method", in *2008 14th IEEE Pacific Rim International Symposium on Dependable Computing*, 2008, pp. 168–175.

[96] J. Xu, Z. Zhang, W. K. Chan, T. H. Tse, and S. Li, "A general noise-reduction framework for fault localization of java programs", *Information and Software Technology*, vol. 55, pp. 880–896, 2013.

[97] L. Naish, H. J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis", *ACM Transactions on Software Engineering and Methodology*, vol. 20, 11:1–11:32, 2011.

[98] L. Naish, H. J. Lee, and K. Ramamohanarao, "Spectral debugging with weights and incremental ranking", in *2009 16th Asia-Pacific Software Engineering Conference*, 2009, pp. 168–175.

[99] W. E. Wong, V. Debroy, Y. Li, and R. Gao, "Software fault localization using dstar (d*)", in *Proceedings of the 2012 IEEE Sixth International Conference on Software Security and Reliability*, ser. SERE 2012, 2012, pp. 21–30.

[100] D. Jeffrey, N. Gupta, and R. Gupta, "Fault localization using value replacement", in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ser. ISSTA 2008, 2008, pp. 167–178.

[101] X. Xie, T. Y. Chen, and B. Xu, "Isolating suspiciousness from spectrum-based fault localization techniques", in *Proceedings of the 2010 10th International Conference on Quality Software*, ser. QSIC 2010, 2010, pp. 385–392.

[102] A. Bandyopadhyay and S. Ghosh, "Proximity based weighting of test cases to improve spectrum based fault localization", in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, 2011, pp. 420–423.

[103] D. Hao, L. Zhang, L. Zhang, J. Sun, and H. Mei, "Vida: Visual interactive debugging", in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE 2009, May 2009, pp. 583–586.

[104] M. D. Weiser, "Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method", Ph.D. dissertation, 1979.

[105] M. Weiser, "Program slicing", in *Proceedings of the 5th International Conference on Software Engineering*, ser. ICSE 1981, 1981, pp. 439–449.

[106] ——, "Program slicing", *IEEE Transactions on Software Engineering*, vol. SE-10, pp. 352–357, 1984.

[107] D. W. Binkley and K. B. Gallagher, "Program slicing", in ser. Advances in Computers, vol. 43, 1996, pp. 1–50.

[108] J.-F. Bergeretti and B. A. Carré, "Information-flow and data-flow analysis of while-programs", *ACM Transactions on Programming Languages and Systems*, vol. 7, pp. 37–61, 1985.

[109] D. Jackson and E. J. Rollins, "Chopping: A generalization of slicing", Tech. Rep., 1994.

[110] R. Gupta and M. L. Soffa, "Hybrid slicing: An approach for refining static slices using dynamic information", in *Proceedings of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering*, ser. FSE 1995, 1995, pp. 29–40.

[111] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs", in *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, ser. PLDI 1988, 1988, pp. 35–46.

[112] L. Larsen and M. J. Harrold, "Slicing object-oriented software", in *Proceedings of the 18th International Conference on Software Engineering*, ser. ICSE 1996, 1996, pp. 495–505.

[113] J. Zhao, "Slicing aspect-oriented software", in *Proceedings of the 10th International Workshop on Program Comprehension*, ser. IWPC 2002, 2002, pp. 251–.

[114] G. A. Venkatesh, "The semantic approach to program slicing", in *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, ser. PLDI 1991, 1991, pp. 107–119.

[115] A. Nishimatsu, M. Jihira, S. Kusumoto, and K. Inoue, "Call-mark slicing: An efficient and economical way of reducing slice", in *Proceedings of the 21st International Conference on Software Engineering*, ser. ICSE 1999, 1999, pp. 422–431.

[116] T. Takada, F. Ohata, and K. Inoue, “Dependence-cache slicing: A program slicing method using lightweight dynamic information”, in *Proceedings of the 10th International Workshop on Program Comprehension*, ser. IWPC 2002, 2002, pp. 169–.

[117] R. J. Hall, “Automatic extraction of executable program subsets by simultaneous dynamic program slicing”, *Automated Software Engineering*, vol. 2, pp. 33–53, 1995.

[118] J. Beck and D. Eichmann, “Program and interface slicing for reverse engineering”, in *Proceedings of the 15th International Conference on Software Engineering*, ser. ICSE 1993, 1993, pp. 509–518.

[119] J. R. Lyle and M. Weiser, “Automatic program bug location by program slicing”, in *Proceedings of the 2nd International Conference on Computers and Applications*, 1987, pp. 877–882.

[120] K. Gallagher, D. Binkley, and M. Harman, “Stop-list slicing”, in *2006 Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, 2006, pp. 11–20.

[121] J. Krinke, “Slicing, chopping, and path conditions with barriers”, *Software Quality Journal*, vol. 12, pp. 339–360, 2004.

[122] G. Canfora, A. Cimitile, and A. D. Lucia, “Conditioned program slicing”, *Information and Software Technology*, vol. 40, pp. 595–607, 1998.

[123] J. Field, G. Ramalingam, and F. Tip, “Parametric program slicing”, in *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL 1995, 1995, pp. 379–392.

[124] C. Fox, M. Harman, R. Hierons, and S. Danicic, “Backward conditioning: A new program specialisation technique and its application to program comprehension”, in *Proceedings 9th International Workshop on Program Comprehension*, 2001, pp. 89–97.

[125] M. Harman, R. Hierons, C. Fox, S. Danicic, and J. Howroyd, “Pre/post conditioned slicing”, in *Proceedings of the IEEE International Conference on Software Maintenance*, ser. ICSM 2001, 2001, pp. 138–.

[126] R. Jhala and R. Majumdar, “Path slicing”, in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2005, 2005, pp. 38–47.

[127] M. Harman and S. Danicic, “Amorphous program slicing”, in *Proceedings of the 5th International Workshop on Program Comprehension*, ser. WPC 1997, 1997, pp. 70–.

[128] M. G. Nanda and S. Ramesh, “Slicing concurrent programs”, in *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2000, 2000, pp. 180–190.

[129] J. Krinke, “Context-sensitive slicing of concurrent programs”, in *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-11, 2003, pp. 178–187.

[130] A. Orso, S. Sinha, and M. J. Harrold, “Incremental slicing based on data-dependences types”, in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, ser. ICSM 2001, 2001, pp. 158–.

[131] J. Hatcliff, M. B. Dwyer, and H. Zheng, “Slicing software for model construction”, *Higher-Order and Symbolic Computation*, vol. 13, pp. 315–353, 2000.

[132] H. Agrawal, J. R. Horgan, E. W. Krauser, and S. London, “Incremental regression testing”, in *Proceedings of the Conference on Software Maintenance*, ser. ICSM 1993, 1993, pp. 348–357.

[133] T. Wang and A. Roychoudhury, “Dynamic slicing on java bytecode traces”, *ACM Transactions on Programming Languages and Systems*, vol. 30, 10:1–10:49, 2008.