2024

Program Analysis For Software Engineers And Students

Jialiang Tan
William & Mary - Arts & Sciences, akunkun820@gmail.com

Program Analysis for Software Engineers and Students

Jialiang Tan

Xi'an, Shaanxi, China

Bachelor of Engineering, Sichuan University, 2015
Master of Computer Science, Arizona State University, 2016
Master of Science, William & Mary, 2019

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

Department of Computer Science

The College of William & Mary in Virginia
January 2024

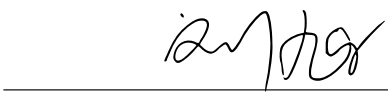
APPROVAL PAGE

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

Doctor of Philosophy

Jialiang Tan

Approved by the Committee, January 2024



Committee Co-chair
Xu Liu, Associate Professor, Computer Science
North Carolina State University

Committee Co-chair
Bin Ren, Associate Professor, Computer Science
College of William & Mary

Weizhen Mao, Professor, Computer Science
College of William & Mary

Andreas Stathopoulos, Professor, Computer Science
College of William & Mary

Jiajia Li, Assistant Professor, Computer Science
North Carolina State University

ABSTRACT

Software inefficiencies are inevitable in computer systems. At the code level, software packages have become increasingly complex, this complexity often introduces inefficiencies across software stacks, leading to performance degradation. At the resource level, the evolution of hardware outpaces the performance optimization of software, leading to resource wastage and energy dissipation in emerging architecture. To better understand program behaviors, software developers take advantage of performance profiling tools. Existing profiling techniques, whether fine-grained profilers or coarse-grained profilers focus on identifying hotspots, however, hotspot analysis hardly diagnoses whether a resource is being used in a productive manner of a program. Thus, developers need to make extra effort to decide if a hotspot needs to be optimized. For this reason, to better perform program optimizations, we need tools that investigate resource wastage rather than resource usage.

In this dissertation, we perform program inefficiency detection from different perspectives. First, we study the inefficiency in compiler optimizations. We propose CIDetector, a fine-grained profiler, to detect compiler-introduced and compiler-missed inefficiencies. Through our analysis, we select 12 representative programs from different domains to form a dataset CIBench. We perform the first study on compiler-related inefficiencies in fully optimized binary codes, it offers valuable insights for scientific programmers, compiler writers, and tool developers. Moreover, we study the interaction inefficiency for Python applications, and extract two inefficiency patterns that are common in interaction inefficiencies. Based on these patterns, we categorize the interaction inefficiencies by their root causes. We propose PieProf, a lightweight profiler, to pinpoint interaction inefficiencies in Python applications. The principle of PieProf is to measure the inefficiencies in the native execution and associate inefficiencies with high-level Python code to provide a holistic view. Guided by PieProf, we optimize 17 real-world applications, yielding speedups up to 6.3× on application level.

In the meantime, we notice the same program inefficiency patterns occur in students' codes. As instructors, we realized that the importance of code performance education to students can never be exaggerated. By exploring the pedagogical method and developing educational tools, we hope to understand and address the challenges that students have during programming. We report our experience of integrating VS Code into an introductory-level Python programming course, together with comprehensive guidance, it significantly balances the teaching resources and shortens the students' learning curves. Additionally, we propose ProTracker, an end-to-end solution to estimate the progress of programming assignments with machine learning techniques. ProTracker employs static analysis to extract features from assignment samples from previous semesters, and applied a two-level cross-validation method for tuning and selecting the proper machine-learning model. It runs as a VS Code extension and performs real-time programming progress estimation for students.

TABLE OF CONTENTS

Acknowledgments ix

Dedication x

List of Tables xi

List of Figures xii

1 Introduction 2

1.1 Contribution Highlights 4

1.1.1 Software Inefficiency Analysis 4

Inefficiency Detection in Compiler Optimizations. 4

Interaction Inefficiency Detection in Python Applications. 4

1.1.2 Computer Science Education Research 5

Advanced Learning Environment. 5

Educational Data Mining. 6

1.2 Dissertation Organization 7

2 What Every Scientific Programmer Should Know About Compiler Optimizations? 8

2.1 Introduction 8

Contribution. 10

2.2 Research Scope and Methodology 11

2.3 CIDetector Design . . . 13
Dead stores. . . . 13
Redundant stores. . . . 14
Redundant loads. . . . 14
2.4 CIBench Design . . . 15
2.4.1 Value Hoisting . . . 16
Hmmer. . . . 16
Srad_v2. . . . 17
2.4.2 Instruction Scheduling . . . 18
Bzip2-1.0.6. . . . 18
2.4.3 Suboptimal Scalar Replacement . . . 19
Hotspot3D. . . . 19
FFT of GSL-2.1.5. . . . 20
MSB-2. . . . 20
2.4.4 Missing Identity Computation and Constant Propagation Optimization . . . 21
MSB1. . . . 21
Backprop. . . . 21
LavaMD. . . . 22
2.4.5 Missing Function Inlining . . . 22
Povray. . . . 23
H264ref. . . . 23
USQCD Chroma. . . . 23
2.4.6 Limited Interprocedural Analysis . . . 24
Hoard. . . . 24
2.5 Experimental Setups and Results . . . 25
2.5.1 Inefficiencies due to Value Hoisting . . . 26

2.5.1.1 Hmmer . . . 26
Vertical study. . . . 28
Horizontal study. . . . 28
2.5.1.2 Srad_v2 . . . 28
Vertical study. . . . 28
Horizontal study. . . . 29
2.5.2 Inefficiencies due to Instruction Scheduling . . . 29
Vertical study. . . . 29
Horizontal study. . . . 29
2.5.3 Missing Scalar Replacement . . . 29
Vertical study. . . . 29
Horizontal study. . . . 30
2.5.4 Missing Function Inlining . . . 30
Vertical study. . . . 30
Horizontal study. . . . 31
2.5.5 Missing Constant Propagation . . . 31
Vertical study. . . . 31
Horizontal study. . . . 32
2.5.6 Missing Interprocedural Optimization . . . 32
2.6 Insights from the Study . . . 33
2.6.1 Insights for Scientific Programmers . . . 33
Inefficiencies exist in fully optimized binary codes. . . . 33
Finding inefficiency patterns in source code is difficult which makes tools necessary. . . . 33
Inefficiencies can be removed by curating source code. . . . 34
Source code optimization does not always yield portable performance. . . . 34

2.6.2 Insights for Compiler Writers . . . 34
Insights from the vertical study. . . . 34
Insights from the horizontal study. . . . 35
2.6.3 Insights for Tool Developers . . . 36
2.7 Related Work . . . 36
2.8 Summary . . . 37

3 Toward Efficient Interactions between Python and Native Libraries 38
3.1 Introduction . . . 38
Contributions. . . . 40
3.2 Background and Related Work . . . 41
3.2.1 Python Runtime System . . . 41
Python basics. . . . 41
Interaction with native libraries. . . . 42
3.2.2 Existing Tools vs. PieProf . . . 43
Python performance analysis tools. . . . 43
Native performance analysis tools. . . . 43
3.2.3 Performance Monitoring Units and Hardware Debug Registers . 44
3.3 Interaction Inefficiency Characterization . . . 44
3.3.1 Interaction Inefficiency Categorization . . . 45
Slice underutilization. . . . 45
Repeated native function calls with the same arguments. 45
Inefficient algorithms. . . . 46
API misuse in native libraries. . . . 47
Loop-invariant computation. . . . 47
3.3.2 Common Patterns in Interaction Inefficiencies . . . 48
3.4 Design and Implementation . . . 49

3.4.1 Overview . . . 49
3.4.2 Measurement . . . 51
CL-algorithm. . . . 51
Improving measurement efficiencies. . . . 52
3.4.3 Calling Context Trees Builder . . . 52
Hybrid call path. . . . 53
CCT from call paths. . . . 54
3.4.4 Safeguard . . . 55
Hibernation at function-level. . . . 55
Blocking garbage collector. . . . 56
3.5 Evaluation . . . 57
3.5.1 Effectiveness . . . 57
3.5.2 Overhead . . . 58
3.6 Threats to Validity . . . 62
3.7 Summary . . . 62

4 Visual Studio Code in Introductory Computer Science Course: An Experience Report 63
4.1 Introduction . . . 63
Contributions. . . . 65
4.2 Background and Related Work . . . 66
4.2.1 Python in Education . . . 66
4.2.2 Pedagogical Approaches . . . 66
4.2.3 IDEs in Education . . . 67
4.3 Visual Studio Code Investigation . . . 68
Accessibility. . . . 68
Easy-to-use. . . . 68

Functionality. 70
Popularity. 70
4.4 Guidance and Support 71
Download and installation of VS Code. 71
Download and installation of Python. 72
Setup Python environment. 72
Run Python examples. 72
Install and run Jupyter extension. 73
Install and use scientific packages. 73
4.4.1 Improvements for Guidance 73
4.4.2 Hierarchical Indexing 74
4.5 Evaluation and Student Responses 75
4.5.1 Course Description 75
4.5.2 VS Code and VS Code Guidance Evaluation 75
4.5.3 Class Averages 77
4.5.4 Student Comments 78
4.5.5 Issues with Jupyter Notebook in VS Code 79
4.6 Discussions 80
Discussions on education-related concerns. 80
Discussions on some limitations. 80
4.7 Summary 80

5 ProTracker: Estimating Progress of Programming Assignments 82
5.1 Introduction 82
Contributions. 85
5.2 Related Works 86
5.2.1 Predictions in Education 86

5.2.2 Progress Monitoring in Education . . . 86
5.3 Design and Implementation . . . 87
5.3.1 Input Dataset . . . 88
5.3.2 ProTracker Backend . . . 89
Preprocessor. . . . 89
Generator. . . . 89
Model selector. . . . 90
5.3.3 ProTracker Frontend . . . 91
5.3.4 Discussion . . . 93
Sequential labeling. . . . 93
Feature selection. . . . 93
Models. . . . 94
5.4 Evaluation . . . 94
5.4.1 Datasets . . . 94
Course description. . . . 94
Assignment description. . . . 95
5.4.2 Accuracy . . . 95
Evaluation setup. . . . 95
Results. . . . 96
5.4.3 Overhead . . . 96
Evaluation setup. . . . 96
Results. . . . 97
5.5 Limitations . . . 98
5.6 Summary . . . 98

6 Conclusion and Future Research Directions 99
6.0.1 Summary of Dissertation Contributions . . . 99

Inefficiency Detection in Compiler Optimizations. 99
Interaction Inefficiency Detection for Python Applications. 100
Computer Science education research. 100
6.0.2 Future Directions 101
Performance profiling on emerging platforms and architectures. 101
Education-related extensions. 101

Bibliography 101

Vita 126

ACKNOWLEDGMENTS

I would like to express my sincere gratitude and appreciation to everyone who helped and supported me in the doctoral journey. First, I would like to thank my advisor, Dr. Xu Liu, for his invaluable academic guidance. His expertise in research, dedication to excellence, and passion for knowledge have shaped me into a better researcher. He has been more than a mentor, I am deeply grateful for his patience, encouragement and belief in me throughout the years of my Ph.D.

Additionally, I would like to thank my defense committee members, Dr. Bin Ren, Dr. Weizhen Mao, Dr. Andreas Stathopoulos, and Dr. Jiajia Li, for their time and valuable feedback to my presentation and dissertation.

I would like to extend thanks to my dear collaborators and friends, Dr. Shuyin Jiao, Dr. Pengfei Su, Yueming Hao, Judy Chen and her family. Thank you for being the wonderful friends that you are.

Lastly, I would like to thank my family. I thank Hangwei Shao, Anbin Tan, and Xiaole Tan, for being my pillars of strength, my source of love, and my lifelong supporters. Their boundless faiths and encouragements have given me the strength to pursue my dreams and reach for the stars. I thank Yu Chen, he has been my companion, my confidant, and my partner in adventure. Special thanks to Lecun, Felicis, Liz, and Yangyang, you presences have brought laughter to my days and added depth to my happiest moments.

To Hangwei and Anbin.

LIST OF TABLES

2.1 The Overview of CIBench dataset. 14
2.2 Optimization capabilities of different compilers. 33

3.1 Redundant loads and stores detect different categories of interaction inefficiencies. 48
3.2 Overview of performance improvement guided by PieProf. *AS* denotes application-level speedup, *FS* denotes function-level speedup, *L* refers to redundant loads and *S* refers to redundant stores. 56

4.1 The averaged answers of students' satisfaction (82 students in total) over four aspects: visual appeal, extension ecosystem, debugging expe-rience, and editing experience. The degree of satisfaction lies between 1 to 5, in which 1 is strongly dissatisfied and 5 is strongly satisfied. . . 77
4.2 The class average of total score in the Spring 2022, Fall 2022, and Spring 2023 semesters. 77

5.1 Topics and learning outcomes of corresponding assignments in our introductory-level Python programming course. 92
5.2 The sample size for six datasets. 94
5.3 Forecast results for each dataset. 95

LIST OF FIGURES

2.1 The overview of CIDetector in the software stack. 12

2.2 Experimental results for Hmmer. The left y-axis denotes speedups and the right y-axis denotes the percentage reduction in dead stores and total stores. The error bars denote the standard deviation across three executions. 26

2.3 Experimental results for programs in CIBench. The left y-axis denotes speedups and the right y-axis denotes the percentage reduction of dead/redundant stores and total stores. The error bars denote the standard deviations across three executions. 27

3.1 The typical stack of production Python software packages. Python applications usually rely on native libraries for high performance but introduce an abstraction across the boundary of Python runtime and native libraries. 42

3.2 Overview of PieProf's workflow. 50

3.3 Constructing a hybrid call path across Python runtime and native libraries. White arrows in call paths denote a series of elided call frames in PVM. The red circle in the hybrid call path shows the boundary of Python and native frames, where interaction inefficiencies occur. 51

3.4 A calling context tree constructed by PieProf. Each parent node applies skip-list to organize children. `INode` denotes an internal node and `LNode` denotes a leaf node. Red box shows searching `0xa46` in the example skip-list. 54

3.5 Runtime slowdown of PieProf on Scikit-learn, Numexpr, and NumpyDL with sampling rates of 500K, 1M, and 5M. The y-axis denotes the slowdown ratio and the x-axis denotes the program name. . 59

3.6 Memory bloating of PieProf on Scikit-learn, Numexpr, and NumpyDL with sampling rates of 500K, 1M, and 5M. The y-axis denotes the slowdown ratio and the x-axis denotes the program name. . 60

4.1 The basic layout of VS Code user interface [32] that we introduced to students. The editor is the area to edit files, students can open multiple editors at the same time side by side vertically or horizontally. The panel below the editor is for output or debug information, errors, and warnings, or an integrated terminal. The side bar contains different views like the Explorer or Extension Marketplace, to assist students while working on the projects or downloading extensions. . . 69

4.2 The red box indicated the Command Palette of VS Code. Students can access all functionality of VS Code and install extensions through the Command Palette. 70

4.3 An example of code completion supported by VS Code, where the IntelliSense suggests using variable `msg` in `print` function. 71

5.1 How we help students in code assignments. We provide test cases to check code correctness, and ProTracker to estimate coding progress. 85

5.2 The overview of ProTracker. It takes historical students' programming assignments as inputs and achieves programming progress prediction through the backend and frontend, which interact with cloud storage. 87
5.3 An example labeled assignment. 88
5.4 An example of real-time programming progress prediction performs by the frontend. The start estimating message and progress estimation appear whenever students save code text. 92
5.5 Overhead of ProTracker's frontend (run as a VS Code extension). 97

Program Analysis for Software Engineers and Students

Chapter 1

Introduction

Software inefficiency analysis is an important research topic in the software engineering community, it is inevitable in computer systems ranging from smartphones to supercomputers and data centers. At the code level, software packages have become increasingly complex. They are comprised of a large amount of source code, sophisticated control and data flow, a hierarchy of component libraries, and growing levels of abstraction. This complexity often introduces inefficiencies across software stacks, leading to performance degradation. At the resource level, the evolution of hardware outpaces the performance optimization of software, leading to resource wastage and energy dissipation in emerging architectures.

Performance profiling abounds in the tools community to aid software developers in understanding their program behavior. Existing profiling techniques, no matter fine-grained ones (e.g., TAU, Scalasca) or coarse-grained ones (e.g., Intel VTune [141], Linux Perf [41], HPCToolKit [2]), focus on identifying hotspots. The hotspot refers to the code region that consumes plenty of resources during program execution. Hotspot analysis is indispensable; however, it hardly diagnoses whether a resource is being used in a productive manner that contributes to the overall efficiency of a program. Hence significant burden is on the developer to make a judgment call on whether there is scope to optimize hotspots. Derived metrics, e.g., Cycles-Per-Instruction (CPI), and cache miss ratio, offer slightly

better intuition into hotspots but are still not panaceas. There is a need for tools that investigate resource wastage instead of resource usage.

In this dissertation, we introduce two research works that perform program inefficiency detection and provide optimization guidance for software packages: (1) CIDetector and CIBench, that perform the first study on compiler-introduced and compiler-missed optimizations in fully optimized binary codes; (2) PieProf, a lightweight profiler that pinpoints interaction inefficiencies between Python codes and native libraries.

With the help of powerful profilers, we extensively examine a wide range of real-world production software packages and influential codebases, at the same time, as teaching assistants and instructors, surprisingly we found the same types of inefficiency in student assignment and project codes. It brings to our attention that programming education in universities needs to focus more on code quality and performance training. How to educate students on programming is always an important topic.

As system researchers, to improve education on code quality and performance, we focus on IDEs (integrated development environments), which play an important role in learning a programming language; with the help of IDEs, student benefit from an efficient development phase, and build better coding habits. We introduce two educational research works that focus on curricular development and innovation: (1) We report the experience of integrating Visual Studio Code in a CS1 Python programming course, which significantly balances the teaching resources and shortens the learning curve; (2) We propose ProTracker, which is a machine learning-based solution to estimate students' real-time programming progress.

By adding new techniques to the renovated pedagogical learning environment, we better understand and address the challenges that students will have during programming, which helps students write high-quality code, further succeeds in their computer science fundamental studies and future careers, and eliminates the occurrence of software inefficiencies from the root.

1.1 Contribution Highlights

My dissertation research falls into the intersection of software inefficiency analysis and computer science education research.

1.1.1 Software Inefficiency Analysis

Inefficiency Detection in Compiler Optimizations. Compilers are an indispensable component in the software stack. Besides generating machine code, compilers perform multiple optimizations to improve code performance. Typically, scientific programmers treat compilers as a blackbox and expect them to optimize code thoroughly. However, optimizing compilers are not a performance panacea. They can miss optimization opportunities or even introduce inefficiencies that are not in the source code. There is a lack of tool infrastructures and datasets that can provide such a study to help understand compiler optimizations.

We investigate an important compiler optimization—dead and redundant operation elimination. We first develop a tool CIDetector to analyze a large number of programs. In our analysis, we select 12 representative programs from different domains to form a dataset called CIBench. We utilize five compilers to optimize CIBench with the highest optimization options available and leverage CIDetector to study each generated binary. We provide insights into two aspects. First, we show that modern compilers miss several optimization opportunities, in fact, they even introduce some inefficiencies, which require programmers to refactor the source code. Second, we show how compilers have advanced in a vertical evolution (the same compiler of different release versions) and a horizontal comparison (different compilers of the most recent releases). With empirical studies, we provide insights for software engineers, compiler writers, and tool developers.

Interaction Inefficiency Detection in Python Applications. Python has become a popular programming language because of its excellent programmability. Many modern

software packages utilize Python for high-level algorithm design and depend on native libraries written in C/C++/Fortran for efficient computation kernels. However, the abstraction lying on the boundary of Python and native libraries introduces performance losses on the interaction between Python code and native libraries, which we refer interaction inefficiencies. Existing efforts cannot address such inefficiencies. On the one side, Python code, typically run with interpretation, is disjoint from its execution behavior. Python profilers [58, 197, 61, 174, 143, 63, 129, 136, 172] fail to step in native code and analyze its execution details. On the other side, native profiling tools [141, 41, 118, 2, 23, 190, 163, 191] locate hotspots, offering insights for inefficiencies diagnosis. However, these tools do not have Python code's semantics to understand algorithm defects, they cannot identify the detailed root cause of the inefficiencies. A comprehensive solution for interaction inefficiencies that bridges the knowledge gap between Python and native code is demanding.

To understand the interaction inefficiencies, we extensively study a large collection of Python software packages and categorize them according to the root causes of inefficiencies. We extract two inefficiency patterns that are common in interaction inefficiencies. Based on these patterns, we develop PieProf, a lightweight profiler, to pinpoint the interaction inefficiencies in Python applications. PieProf applies CL-algorithm[1] [191, 162], an efficient redundant memory access detecting method, and leverages it in a complicated multi-languages environment, in which Python is used to control the semantics and native libraries are used to perform high-performance computation. The principle of PieProf is to measure the inefficiencies in the native execution and associate inefficiencies with high-level Python code to provide a holistic view. Guided by PieProf, we optimize 17 real-world applications, yielding speedups up to 6.3× on the application level.

1.1.2 Computer Science Education Research

Advanced Learning Environment. Involving integrated development environments (IDEs) in introductory-level (CS1) programming courses is critical. IDEs play an impor-

[1] Chabbi-Liu Algorithm.

tant role in learning a programming language; through the help of IDEs, students benefit from efficient programming, testing, and debugging; students further develop better coding habits and flatten the learning curve of a new language. As a result, more and more instructors start involving IDEs in introductory-level Python courses [131, 3, 180], which significantly improves students' coding experiences. However, it is difficult for instructors to find a suitable IDE that is beginner friendly and supports strong functionality.

In this work, we report the experience of integrating Visual Studio Code (VS Code) in a CS1 Python programming course at the Department of Computer Science, North Carolina State University. We describe our motivation for choosing VS Code and how we introduce it to students. We create comprehensive guidance with hierarchical indexing to help students with diverse programming backgrounds, which significantly balances the teaching resources and shortens the learning curve. We perform an experimental evaluation of students' programming experience of using VS Code and validate the VS Code together with guidance as a promising solution for CS1 programming courses.

Educational Data Mining. Tracking the progress of programming assignments gives useful feedback to students and instructors to better manage time and teaching resources. However, there is no such tool available in the programming environment for students and instructors to easily understand the progress accurately and efficiently.

We propose ProTracker, a machine learning-based solution to estimate the progress of programming assignments in real time. ProTracker generates datasets by extracting features in assignment samples from previous semesters, applies a two-level cross-validation method for tuning and selecting appropriate machine-learning models. The frontend of ProTracker runs as a VS Code extension, Based on the current codes (in the text format) in students' editors. We evaluate ProTracker on six datasets and observe that ProTracker achieves an average R^2 score of 0.86 with 2.5ms overhead for real-time estimating programming progress of assignments.

1.2 Dissertation Organization

The remainder of this dissertation is organized as follows. In Chapter 2, we propose CIDetector, a whole-program fine-grained profiler that works on fully optimized binaries, and CIBench, the first dataset with 12 representative programs from various domains for studying compiler-related inefficiencies. In Chapter 3, we present PieProf, a profiler to identify interaction inefficiencies and provide intuitive optimization guidance. In Chapter 4, we report the experience of integrating VS Code and VS Code Guidance in a CS1 Python programming course. In Chapter 5, we introduce ProTracker, an end-to-end solution to estimate the real-time progress of programming assignments with machine learning techniques. And in Chapter 5, we describe the conclusion and discuss future research directions.

Chapter 2

What Every Scientific Programmer Should Know About Compiler Optimizations?

2.1 Introduction

Compilers are an important component in the modern software stack. They, typically, consume source code as inputs, generate machine code[1], and apply various optimizations. In practice, scientific programmers typically treat compiler as a blackbox and expect it to do a good job at a high optimization level (e.g., `-O3` for most compilers) for performance. However, optimizing compilers are not a panacea; it is challenging for compilers to deliver "bare-metal" performance due to multiple reasons.

Production software packages of various domains have become increasingly complex; they integrate multiple components for various functionalities, and employ sophisticated flow of control and a hierarchy of component libraries. This complexity often introduces inefficiencies in the software stacks, which creates challenges for optimizing compilers. Optimizing compilers are adept at eliminating inefficiencies with techniques such as common

[1]Some compilers are also used in source-to-source translation.

subexpression elimination [42], value numbering [144], constant propagation [189], partial redundancy elimination [171], dead code elimination [171], among others. Although profile-guided optimization (PGO) [168] and link-time optimization (LTO) [55, 86, 80] can further optimize code with extra information such as runtime statistics and better visibility in the entire library hierarchy, the myopic view of the program yields conservative gains.

Thus, there is an urge from performance sensitive developers, such as the scientific programmers in the High-Performance Computing (HPC) domain, to understand the limitations in popular compilers and avoid performance pitfalls associated with optimizing compilers and achieve better runtime performance by curating their code.

This paper bridges the gap between scientific programmers and compilers. In this paper we hope to answer the following questions:

1. Which inefficiencies compilers introduce or fail to optimize?
2. Why optimizing compilers fail to eliminate inefficiencies?
3. Are compilers of different versions or different vendors similar in optimizing inefficiencies?
4. How can programmers become aware of inefficiencies that are not optimized by compilers? Do common patterns exist?
5. Can source code modification eliminate inefficiencies? If yes, how many code lines does one typically need to change?
6. Can source code optimization always yield portable speedups for code compiled by different compilers?

A major challenge of this research is the lack of standard datasets for experiments. Although some prior studies [117, 153, 23, 190, 164, 46] identify inefficient computations, they are not focused on compiler optimizations. Instead, many inefficiencies studied in the

prior work are related to algorithms, data structures, and inputs. Some prior works [83, 95, 150] text mine commit logs in public repositories; these techniques, typically, do not reveal any inefficiencies arising from compiler-introduced problems and optimizations missed by compilers, which is the focus of our work.

We take a step in addressing these challenges by developing a tool infrastructure (CIDetector [2]) to analyze a large number of software programs and identify interesting ones to form the dataset (CIBench) for the study. CIDetector can (1) pinpoint inefficiencies to guide code optimization as all the programs in the dataset CIBench have both original and our hand-optimized versions, and (2) quantify the inefficiencies in the machine code produced by various compilers for our study. We hope the community will adopt CIBench, add more coverage to make it comprehensive, and inspire compiler writers to develop optimization techniques[3] to match or beat the performance of our hand optimizations.

With the help of CIDetector and CIBench, we attempt to answer the aforementioned six questions and provide insights for programmers, compiler writers, and tool developers.

Contribution. In this work, we make the following contributions.

- We develop CIDetector, a tool to analyze a large number of applications and pinpoint dead/redundant operations in fully optimized binary executables generated by a variety of compilers.
- We select 12 representative code bases with the help of CIDetector to form the dataset CIBench, which is the first dataset for studying compiler-related inefficiency. We characterize, optimize, and categorize CIBench in multiple ways.
- We leverage CIDetector and CIBench to study the implications of compiler evolution on inefficiency optimization, not only with the same compiler of different releases

[2] CI denotes compiler inefficiency.
[3] including adopting optimizations discussed in the rich compiler literature

but also with different state-of-the-art compilers.

- We provide analytical insights as well as open source CIDetector and CIBench to benefit scientific programmers, compiler writers, and tool developers.

2.2 Research Scope and Methodology

Assessing all programs compiled through all compilers on all architectures is simply infeasible. We need to make pragmatic choices to scope our evaluation. First, we limit our study to scientific programs, where compilers play a crucial role in delivering high performance; moreover, scientific computing is a field that is at the forefront of performance demands.

Second, we choose to observe inefficiencies at the machine-code level, which can expose the inefficiencies not shown up in the source/intermediate code. However, we attribute our observations back to the code and program constructs for optimization. Furthermore, for pragmatic reasons, we evaluate only `x86_64`, which is the most widely deployed architecture.

Third, we choose dynamic program analysis, which can overcome the limitations in static analysis, such as imprecise alias and pointer analysis. Actually, we assume the binary code under investigation is thoroughly optimized by various static analysis techniques in compilers. This choice means we can only observe the code that executes on our inputs.

Finally, with thousands of compiler optimizations developed over more than several decades of research, it would be impractical to identify whether each one of them is applied or missed at runtime. We focus on a particular class of compiler optimization — elimination of useless operation. A vast number of optimizations fall in this category, e.g., dead code elimination, load elimination, constant propagation, lazy code motion, loop invariant code motion, to name a few. Hence, our choice of looking for useless operations at runtime provides good coverage, as it will become evident in our evaluation.

We define three kinds of useless operations in the context of our dynamic analysis for

detecting compiler inefficiencies:

Definition 1 *Dead Store. A dead store occurs when two consecutive store operations to the same memory location are not intervened by a load operation on the same location.*

Definition 2 *Redundant Store (a.k.a Silent Store). A store operation, writing a value V at register/memory location M, is redundant (or silent) if M already holds the same value V.*

Definition 3 *Redundant Load. A load operation, loading a value V from memory location M, is redundant if the immediately preceding load from M also loaded the same value V.*

Our choice of these inefficiency detection techniques may seem narrowly scoped; but, as you will notice in our evaluation, they are very powerful in detecting missed-optimizations as well as compiler-introduced inefficiencies. For example, a missed scalar replacement will appear as repeated loads of the same data from the same location within or across loop iterations; a missed inlining may appear as storing the same values to the same stack slots (arguments) in the same calling context. Code hoisting onto hot paths will appear as operations whose values are never used before being overwritten.

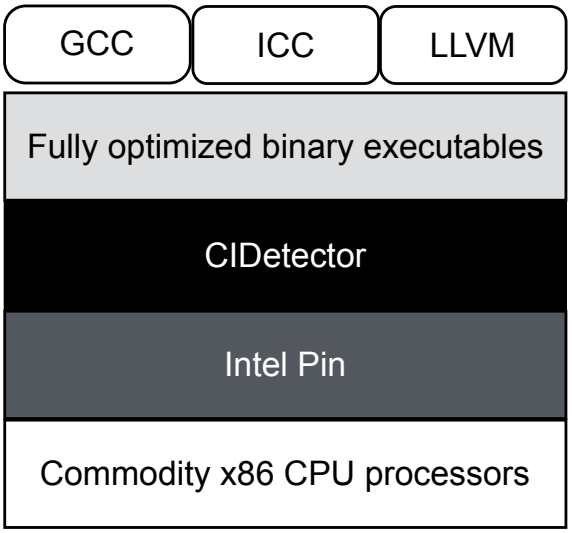


Figure 2.1: The overview of CIDetector in the software stack.

2.3 CIDetector Design

CIDetector has two functionalities. First, with CIDetector, we can analyze a large number of programs and select the ones suitable for CIBench. Second, with CIDetector, we are able to analyze programs in CIBench that are optimized by different compilers. CIDetector should be compiler-independent and provide deep insights. Figure 2.1 provides an overview of the position of CIDetector in the software stack. CIDetector is built atop Intel Pin [97, 31] dynamic binary instrumentation tool and accepts fully optimized executable. CIDetector leverages Pin to instrument memory and register accesses (including inspecting the runtime values) and detects useless operations as defined in Section 2.2.

Dead stores. The driving principle for dead store detection is the invariant that two writes to the same memory location without an intervening read operation makes the first write to that memory location *dead*. To identify dead writes throughout an execution, we monitor every memory read and write operation issued during program execution. For each addressable unit of memory `M` accessed by the program, we assign a state indicating whether the last operation on `M` was a read or a write. If an instruction writes to an address with a write state, a dead store occurs[4]. When a dead store is detected, information is recorded for later reporting. Every dead write has two calling contexts involved — the first write (i.e. the dead write) at $ctxt_d$ and the second write (i.e. the killing write) at $ctxt_k$, represented by the tuple $T_i = (ctxt_d, ctxt_k)$. We maintain a map $T_i : C_i$, where C_i is the number of instances of dead writes observed at the same tuple T_i. CIDetector reports each C_i in the descending order of C_i value, which highlights the top places of dead (and killing) writes in the code. This methodology follows the strategy devised by Chabbi and Mellor-Crummey [23].

[4] A final write to an address without a subsequent read qualifies as a dead store.

Table 2.1: The Overview of CIBench dataset.

Inefficiency Type	Program Name	Cause	Inefficiency Percentage	Problematic Code	Total LoC	Modified LoC	Speedup	Domain
Dead Store	Hmmer [71]	VH[1]	28.9%	fast_algorithms.c:134-150	35992	18	2.28×	Bioinformatics
	Srad_v2 [24]	VH	13.0%	srad.cpp:153-156	239	4	1.11×	Image Processing
	Bzip2-1.0.6 [151]	IS[2]	22.3%	blocksort.c:345-470	8117	15	1.08×	System Software
Redundant Store	LavaMD [24]	IC[3]	4.5%	kernel_cpu.c:173	826	34	1.91×	Scientific Computing
	Backprop [24]	IC	0.6%	backprop.c:323	679	2	1.22×	Machine Learning
Redundant Load	Hotspot3D [24]	SSR[4]	46.8%	3D.c:110,175	272	17	1.46×	Circuit Design
	H264ref [71]	MFI[5]	58.5%	mv-search.c:419	51578	62	1.29×	Multimedia Processing
	Hoard [15]	LIA[6]	67.3% *	libhoard.cpp:127	44730	4	1.50× *	System Software
	NERSC MSB1 [139]	MCP[7]	98.1%	msgrate.c:67	3551	2	3.06×	Network Communication
	NERSC MSB2 [139]	SSR	98.1%	msgrate.c:66	3551	2	1.17× +	Network Communication
	GSL FFT [65]	SSR	68.7%	c_radix2.c:133,134	400475	8	1.03×	Scientific Computing
	Povray [71]	MFI	75.0%	csg.cpp:250	155177	19	1.04×	Image Processing
	USQCD Chroma [53]	MFI	15.8%	qdp_random.h:56	973266	10	1.09×	Scientific Computing

[1]: Value hoisting.
[2]: Instruction Scheduling.
[3]: Identity computation.
[4]: Suboptimal scalar replacement.
[5]: Missing function inlining.
[6]: Limited interprocedural analysis.
[7]: Missing constant propagation.
*: Hoard cannot be compiled by `gcc 4.1.2`, so we report both inefficiency percentage and speedup compiled with `gcc 6.2`.
+: The speedup of MSB2 is a further optimization atop MSB1.

Redundant stores. The driving principle for redundant store detection is the invariant that two writes to the same memory location with the same value makes the second write to that memory location *redundant*. To identify redundant stores, we insert instrumentation before and after a memory write operation. We then need to capture the effective address `e` and the value `v'` at `e` before an instruction's execution into a buffer, say `b`. With this information, the analysis performed immediately after the instruction can compare the new value `v` at `e` with the previous value `v'` captured in `b` for the number of bytes that the instruction writes and flag redundancy *iff* `v=v'`. The mechanism of reporting redundant stores is similar to CIDetector for dead stores, which quantifies the occurrence count of redundant stores and reports the top pairs for optimization. This follows the strategy in [190].

Redundant loads. This is similar to detecting redundant stores: instead of monitoring memory write operations, CIDetector monitors memory reads [163, 164].

CIDetector also associates the calling contexts with source code for intuitive optimization guidance. For studying programs in CIBench, CIDetector computes the inefficiency percentage as the inefficiency instances over the total number of loads (for redundant loads) or stores (for dead and redundant stores).

2.4 CIBench Design

We run CIDetector on a large number (more than 100) of scientific codes to identify the ones that suffer from significant dead/redundant operations (i.e., programs typically suffering from more than 10% redundant stores or 50% redundant loads). These programs include benchmarks from SPEC CPU2006 [71], CPU2017 [21], OMP2012 [108], Rodinia [24], CORAL2 [33], Triniti [139] benchmark suites, as well as some simulation packages from national laboratories, such as USQCD [53] and NWChem [176], and also supporting libraries. e.g., GNU Scientific Library (GSL) [65] and Hoard [15]. We use `gcc 4.1.2 -O3`, the default compiler in Red Hat Enterprise Linux 5 as the baseline compiler[5]. We intentionally choose an older compiler version as our baseline to help assess the evolution. Our approach results in 12 programs with inefficiencies of different reasons to form the dataset CIBench for further analysis.

The design of CIBench follows four criteria:

1. Programs should have compiler-related inefficiencies. We filter out programs with dead or redundant operations due to misuse of data structures, suboptimal algorithms, or skewed inputs.

2. Programs should have significant inefficiencies that are actionable for optimization. As for a self-contained dataset, we provide both original and optimized code for each program in CIBench. The optimization typically yields nontrivial speedups.

3. Programs should be representative of different categories. We select CIBench programs according to different inefficiency categories defined in Section 2.2, and we also cover both compiler-missed optimizations and compiler-introduced inefficiencies.

4. CIBench should cover different domains in scientific computing. We select high-performance computing packages, machine learning programs, and system software

[5]Note that we use other compilers and newer versions in the subsequent sections.

```
1 ▶ mc[k] = mpp[k-1] + tpmm[k-1];
2   if ((sc = ip[k-1] + tpim[k-1]) > mc[k])
3 ▶   mc[k] = sc;
```

Listing 2.1: Dead stores in Hmmer.

```
1  add   (%rbx,%rcx,4),%eax     #hoist the computation result in %eax
2 ▶mov  %eax,0x4(%rsi)         #the 1st store to mc[k]
3  ...
4  cmp   %edx,%eax              #the conditional check uses mc[k]'s value stored in %
       eax instead of 0x4(%rsi)
5  ...
6 ▶mov   %edx,0x4(%rsi)        #the 2nd store to mc[k]
```

Listing 2.2: The assembly code of Listing 2.1.

libraries supporting the computation. Some of them are from well-known benchmark suites and some others are real applications.

Table 2.1 shows all the programs in CIBench categorized by different inefficiency types. We highlight the inefficient code location and quantify the speedups after optimizing the inefficiencies. As the inefficiencies themselves and their optimizations are not straightforward, in this section, we elaborate on these inefficiencies and categorize them according to their root causes.

It is worth noting that some inefficiencies do not show up in the source code; the compilers introduce them during the code generation due to either the conservative optimization (Section 2.4.1) or the aggressive optimization (Section 2.4.2). Other inefficiencies are expected to be optimized by compilers but we observe that they remain unoptimized. We group programs according to the failure of different compiler optimizations in Section 2.4.3–2.4.6.

2.4.1 Value Hoisting

Hmmer. Hmmer is a SPEC CPU2006 benchmark [71]. It uses profile hidden Markov models of multiple sequence alignments to search for patterns in DNA sequences.

Listing 2.1 shows the problematic code piece in Hmmer where many dead stores occur. This code appears in a two-level nested loop, and CIDetector reports that the store

```
1 int mcTmp = mpp[k-1] + tpmm[k-1];
2 if ((sc = ip[k-1]  + tpim[k-1]) > mcTmp)
3  mcTmp = sc;
4 mc[k] = mcTmp;
```

Listing 2.3: Optimized Hmmer that avoids dead stores.

```
1 ▶ c[k] = 1.0 / (1.0+den) ;
2   if (c[k] < 0) {c[k] = 0;}
3 ▶ else if (c[k] > 1) {c[k] = 1;}
```

Listing 2.4: Dead stores in Srad_v2.

in line 3 overwrote the store in line 1. In the unoptimized code, the two stores to mc[k] one each on line 1 and line 3 are separated by a load in the conditional expression on line 2, thus making them non-dead. In the compiler-optimized assembly code as shown in Listing 2.2, the value of `mc[k]` computed on line 1 is held in a register, which is reused during the comparison on line 2; however, the write to memory on line 1 is not eliminated, since the compiler cannot guarantee that the arrays `ip`, `tpim`, and `mc` do not alias each other. Thus in the compiler optimized code, if line 3 executes, it overwrites the previous store to `mc[k]` on line 1.

Optimization. We apply two optimization techniques to eliminate dead stores in Hmmer. First, we use a local variable to record the value and assign its value to `mc[k]` when the condition satisfies, as shown in Listing 2.3. The other optimization is to use "restrict" keyword for all three arrays to inform compilers that they never alias to each other.

Srad_v2. Srad_v2 is a Rodinia benchmark [24], which applies partial differential equations to filter noise in images. Listing 2.4 shows the inefficient code that suffers from a similar problem (dead stores) as Hmmer: the value computed for `c[k]` at line 1 is stored in a register and reused in the condition check at line 2. Unlike Hmmer, this inefficiency only involves one array, which provides a different insight into compilers from handling multiple arrays.

Optimization. Similar to Hmmer, we use a local variable to record the value for `c[k]` and only assign `c[k]` when the condition satisfies.

```
1 Bool mainGtU (UInt32 i1, UInt32 i2, UChar* block, ...){
2     ...
3     /* 1 */
4     c1 = block[i1];
5     c2 = block[i2];
6     if (c1 != c2) return (c1 > c2);
7     i1++; i2++;
8     /* 2 */
9     c1 = block[i1];
10    c2 = block[i2];
11    if (c1 != c2) return (c1 > c2);
12    i1++; i2++;
13    /* 3 */
14    c1 = block[i1];
15    c2 = block[i2];
16    if (c1 != c2) return (c1 > c2);
17    i1++; i2++;
18    ...
19 }
```

Listing 2.5: Dead stores in Bzip2-1.0.6.

2.4.2 Instruction Scheduling

Bzip2-1.0.6. Bzip2 [151] is a tool to compress files. An inlined function `mainGtU` accounts for many dead stores, as shown in Listing 2.5. This function has 12 successive conditions checking the accesses of array `block[i1]` and `block[i2]` through `block[i1+11]` and `block[i2+11]`; this function returns when any check fails. The source code does not show up any dead stores.

However, when looking into the corresponding assembly code in Listing 2.6, we can see there are dead stores due to compiler's instruction scheduling. As shown in Listing 2.6, the compiler aggregates address computation `&block[0]` till `&block[i1 +11]` before the first conditional check (line 6 in Listing 2.5). However, there are not enough registers to store all the pre-computed addresses, hence, they are spilled to the memory during scalar replacement.

For the workload under study, execution of `mainGtU` returns in the early part of the checks, which means in such a cases many of the the pre-computed addresses remain unused; when the spill locations are later overwritten, dead stores show up.

Optimization. We add complex control sequence — a switch statement before conditional checks – to defeat compiler from performing its aggressive code hoisting [51].

```
1 lea     (%r11,%rcx,1),%r8d  #%r8d contains i1
2 lea     0x1(%r8),%ebx       #compute block[i1+1] in %ebx
3 lea     0x2(%r8),%r9d       #compute block[i1+2] in %r9d
4 lea     0x3(%r8),%r15d      #compute block[i1+3] in %r15d
5 ...
6 mov     %rbx,0x168(%rsp)    #spill block[i1+1] to stack
7 mov     %r9,0x160(%rsp)     #spill block[i1+2] to stack
8 mov     %r15,0x158(%rsp)    #spill block[i1+3] to stack
9 ...
10 cmp    %al,(%r15,%rdx,1)   #fist check if(c1 != c2)
11 ...
12 cmp    %al,(%r15,%rdx,1)   #second check if(c1 != c2)
13 ...
```

Listing 2.6: The assembly code of Listing 2.5.

2.4.3 Suboptimal Scalar Replacement

Three programs from CIBENCH suffer from suboptimal scalar replacement but for different reasons. Hotspot3D fails to store array values that are reused across loops into registers; FFT from GSL fails to store reused array values in the same loop iteration into the registers; MSB from NERSC has global constant variables that do not reside in registers. All of these three programs have a large fraction of redundant loads because they frequently load the same values from the same memory location.

Hotspot3D. Hotspot3D from Rodinia Benchmark Suite [24], is a thermal simulation tool, estimating processor temperature based on an architectural floorplan and simulated power measurements [74]. Listing 2.7 shows the problematic code at line 7, where `tOut_t[c]` is updated with the values of `tIn_t[]`. We observe that when `w = c - 1` and `e = c + 1`, the value of `tIn_t[e]` of the current iteration is calculated by `tIn_t[c]` from next iteration and `tIn_t[w]` from the one after next iteration, but the compiler fails to perform register promotion for `tIn_t[e]` in this computation.

Optimization. we manually store the value of `tIn_t[e]` in a local variable for later reuses, aka scalar replacement. The compiler automatically places this local variable into a register.

```
1 for(y = 0; y < ny; y++){
2   for(x = 0; x < nx; x++){
3         ...
4         w = (x == 0)    ? c : c - 1;
5         e = (x == nx-1) ? c : c + 1;
6         ...
7 ▶       tOut_t[c]=cc*tIn_t[c]+cw*tIn_t[w]+ce*tIn_t[e]+ ...
8     }
9 }
```

Listing 2.7: Redundant loads in Hotspot3D.

```
1  for(...){
2     ...
3     REAL(data,stride,j) = REAL(data,stride,i) - wd_real;
4     IMAG(data,stride,j) = IMAG(data,stride,i) - wd_imag;
5     REAL(data,stride,i) += wd_real;
6     IMAG(data,stride,i) += wd_imag;
7 }
```

Listing 2.8: Redundant loads in GSL FFT.

FFT of GSL-2.1.5. The GNU Scientific Library (GSL) [65] is a numerical library for C and C++ programmers, which provides a number of mathematical routines. FFT (Fast Fourier Transformations) is one of such routines.

Listing 2.8 shows the problematic code where `data_real[i]` (i.e., `REAL(data, stride, i)` are loaded twice at line 3 and 5, and `data_imag[i]` (i.e., `IMAG(data, stride, i)`) are loaded twice at line 4 and 6; Both `data_real[i]` and `data_imag[i]` have immutable values in the same iteration and can be stored in registers for reuse. However, due to the unresolved aliases, compilers fail to promote the values into registers.

Optimization. From the premise of never alias, we optimize the code with the scalar replacement. We manually place `REAL(data, stride, i)` and `IMAG(data, stride, i)` in local variables for computation, so as the compiler can place the values into registers.

MSB-2. MSB, a NERSC-8/Trinity Benchmark [139], measures the message passing rate. In Listing 2.9 we find two different types of inefficiencies, for the convenience of discussion, we name the inefficiency described here as MSB2, and the other type of inefficiency described in section 2.4.4 as MSB1. Listing 2.9 highlights the inefficient code at line 8, where the value of variable `cache_size` is repeatedly loaded from memory. The compiler

```
1 int cache_size = (8 * 1024 * 1024 / sizeof(int));
2 ...
3 int *cache_buf;
4 ...
5 static void cache_invalidate(void){
6     int i;
7     cache_buf[0] = 1;
8     for(i = 1; i < cache_size; ++i){
9       cache_buf[i] = cache_buf[i-1];
10    }}
```

Listing 2.9: Redundant loads in MSB.

does not promote this value into a register because `cache_size` is a global variable and the compiler conservatively assumes its value could be modified.

Optimization. We define a local variable for `cache_size` to ensure the register can hold this value and use it safely in the loop.

2.4.4 Missing Identity Computation and Constant Propagation Optimization

Inefficiencies of this category are due to the failure of compiler optimization on computation whose values can be either reasoned about statically or bypassed by inserting conditional checks.

MSB1. MSB has another inefficiency in Listing 2.9. Line 9 sets all `cache_buf` elements to be 1. However, the compiler does not perform the constant propagation to eliminate this loop, possibly due to the inability of the compiler to prove the safety of the assignment of a global array without the awareness of execution.

Optimization. We directly assign 1 to each element of `cache_buf`.

Backprop. Backprop [24], a Rodinia benchmark, trains the weights of connecting nodes on a layered neural network. Listing 2.10 shows the inefficient code, which repeatedly writes the same values to `new_dw` at line 3. The loop shown in Listing 2.10 has been accessed twice: for the first time, the loop iterates 17 times and during iterations the value `new_dw` is non-zero. However, during the second time, the loop has huge iterations

```
1 for(j = i; j <= ndelta; j++){
2     for (k = 0; k <= nly; k++) {
3 ▶       new_dw=((ETA*delta[j]*ly[k])+(MOMENTUM*oldw[k][j]));
4         w[k][j] += new_dw;
5         oldw[k][j] = new_dw;
6     }}
```

Listing 2.10: Redundant stores in Backprop.

```
1 for (i=0; i<NUMBER_PAR_PER_BOX; i=i+1){
2     for (j=0; j<NUMBER_PAR_PER_BOX; j=j+1){
3 ▶       r2 = rA[i].v + rB[j].v - DOT(rA[i],rB[j]);
4         u2 = a2*r2;
5         vij= exp(-u2);
6         fs = 2.*vij;
7         ...}}
```

Listing 2.11: Redundant stores in LavaMD.

of over one million but the value `new_dw` is zero all the time, where the redundancy occurs since adding zero does not change the value of `w`.

Optimization. This compiler-missed optimization can be addressed by adding a conditional check. We only update the value of `w` when `new_dw` is non-zero.

LavaMD. LavaMD, a benchmark from Rodinia Benchmark Suite, calculates particle potential and relocation due to mutual forces between particles within a large 3D space [24]. Listing 2.11 shows the inefficient code, where `exp()` often returns the same value. Further investigation shows that `r2` in line 3 is often assigned with the same value, and `a2` is a loop invariant. As a result, the expensive exponential computation often produces the same value in the loop.

Optimization. We insert a conditional check before line 4. If the value of `r2` keeps unchanged, we reuse the result of `vij` from the previous iteration.

2.4.5 Missing Function Inlining

Inlining small functions with high invocation frequency can significantly improve performance. Failing to inline such functions can show many redundant loads because of frequent accesses to the same activation record on the stack. CIBench has three programs belonging to this category but with different features. Povray has a small function in a separate

```
1 for (Current_Sib = ((CSG *)Object)->Children; Current_Sib != NULL; Current_Sib =
      Current_Sib->Sibling){
2     if ( TEST_RAY_FLAGS(Current_Sib) ){
3 ▶         if (Ray_In_Bound (Ray, Current_Sib->Bound)){
4             ...}}
5 }
```

Listing 2.12: Redundant loads in Povray.

file that is not inlined; H264ref has function pointers that prevent compilers from inlining; Chroma has a function in an external library that is not inlined.

Povray. Povray is a Ray-Tracer from SPEC CPU 2006 Benchmark Suite [71]. CIDetector points out that the major redundant loads occur at line 3 in Listing 2.12. The redundant loads are caused by the invocation of the function `Ray_In_Bound`, which frequently loads the same values to a register to pass the function arguments.

Optimization. We manually inline the *Ray_In_Bound* function to its problematic call site.

H264ref. H264ref, a SPEC CPU2006 benchmark [71], is a reference implementation of H.264/AVC, the latest video compression standard. Listing 2.13 shows the problematic code snippet. The function pointer `PelYline_11` is assigned to `Fastline16Y_11` or `UMVLine16Y_11`, these two functions accept `abs_x`, `img_width`, and `img_height` as arguments, which are loop invariants in the two-level loop nest. As a result, the same values in line 10 are loaded for both callees leading to a great number of redundant loads. Furthermore, due to the overwritten of the same values, there are a significant number of redundant stores as well.

Optimization. We manually inline the function calls.

USQCD Chroma. Chroma [53] is a toolbox that supports data-parallel programming constructs for lattice field theory and in particular lattice quantum chromodynamics (QCD).

Through observation, the majority of redundancies occur in a specific function `sranf`

```
1 for (pos = 0; pos < max_pos; pos++) {
2   ...
3   if (...)
4   PelYline_11 = FastLine16Y_11;
5   else PelYline_11 = UMVLine16Y_11;
6   ...
7   for (blky = 0; blky < 4; blky++){
8    LineSadBlk0 = LineSadBlk1 = LineSadBlk2 = LineSadBlk3 = 0;
9    for (y = 0; y < 4; y++){
10 ▶   refptr=PelYline_11(ref_pic,abs_y++,abs_x,img_height,img_width)
11   ...} ...}}
```

Listing 2.13: Redundant loads in H264ref.

```
1 template<class T1, class T2>
2 inline void
3 fill_random(float& d,T1& seed,T2& skewed_seed,const T1& seed_mult)
4 ▶   d = float(RNG::sranf(seed, skewed_seed, seed_mult));
```

Listing 2.14: Redundant loads in Chroma.

as shown at line 4 in Listing 2.14. The problem is that function `sranf` will keep popping the same values repeatedly from the same stack location to restore the register values. The compiler does not inline `sranf` because this function's definition and invocation are from different libraries.

Optimization. we manually inline `sranf` to all its call sites.

2.4.6 Limited Interprocedural Analysis

Hoard. Hoard [15], a high-performance cross-platform C++ based memory allocator, has been integrated into an array of applications and programming languages such as GNU Bayonne and Cilk programming language. It has 20K lines of code and is parallelized with the `pthreads` library. Hoard's built-in benchmark Larson has a large number of redundant loads. The top redundancy pair is associated with lines 4 and 7 as shown in Listing 2.15.

The cause of such redundancy is that the program repeatedly checks whether `theTLAB` is a null pointer. More specifically, the function `isCustomHeapInitialized` at line 15 and function `getCustomHeap` at line 16 both include code to check whether `theTLAB` is equal to `nullptr`. Hence, the second check at lines 8-11 in `getCustomHeap` is redundant. The compiler does not perform the optimization due to the limited interprocedural analysis.

Optimization. To remove redundancies, we inline these two functions into their caller

```
1 static __thread TheCustomHeapType * theTLAB INITIAL_EXEC_ATTR = nullptr;
2 ...
3 bool isCustomHeapInitialized() {
4 ▶ return (theTLAB != nullptr);
5 }
6 TheCustomHeapType * getCustomHeap() {
7 ▶ auto tlab = theTLAB;
8   if (tlab == nullptr) {
9     tlab = initializeCustomHeap();
10    theTLAB = tlab;
11  }
12  return tlab;
13 }
14 void * xxmalloc (size_t sz) {
15  if (isCustomHeapInitialized()) {
16   void * ptr = getCustomHeap()->malloc(sz);
17   ...
18  }}
```

Listing 2.15: Redundant loads in Hoard.

`xxmalloc` and remove the redundant check.

2.5 Experimental Setups and Results

We conduct our evaluation on a 14-core Intel Xeon E7-4830 v4 machine clocked at 2GHz. The memory hierarchy consists of 32KB and 256KB L1 and L2 private caches, a 30MB L3 shared cache and a 256GB main memory. The operating system is Linux 3.10.

We perform both vertical and horizontal studies of different compiler versions. The vertical study uses `gcc` [122] of different versions (`4.1.2`, `6.2.0`, and `9.3.0`), while the horizontal study uses the latest `gcc 9.3.0`, `icc 19.1` [79], and `llvm 9.0` [92]. We build `gcc` and `llvm` from the source code tree while installing `icc` from the official binary. We enable the highest optimization levels of all these compilers for experiments, including `-O3 -Ofast -march=native -mtune=native`, profile guided optimization (PGO), and link time optimization (LTO) upon the availability. PGO offers runtime statistics and LTO offers whole-program visibility, which provides unique information for optimization. The exception is `gcc 4.1.2`, we use `-O3` as the baseline because it does not support `-Ofast`. We use `gcc 4.1.2` and `6.2.0` for vertical study because they were default releases in popiular Red Hat Enterprise Linux 5 and Red Hat Developer Toolset 6, respectively.

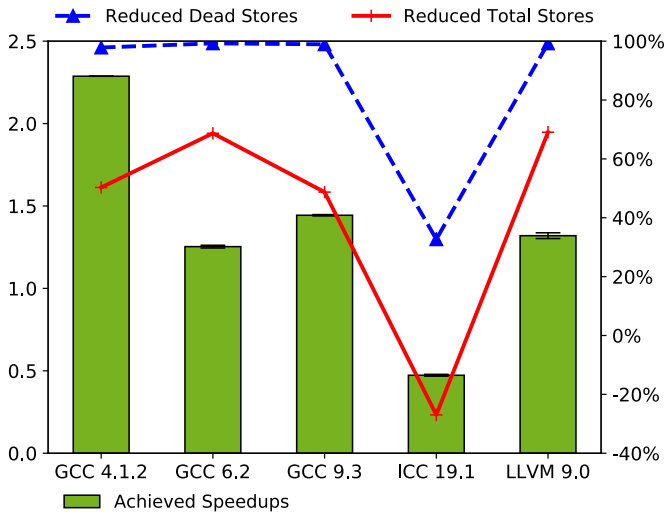


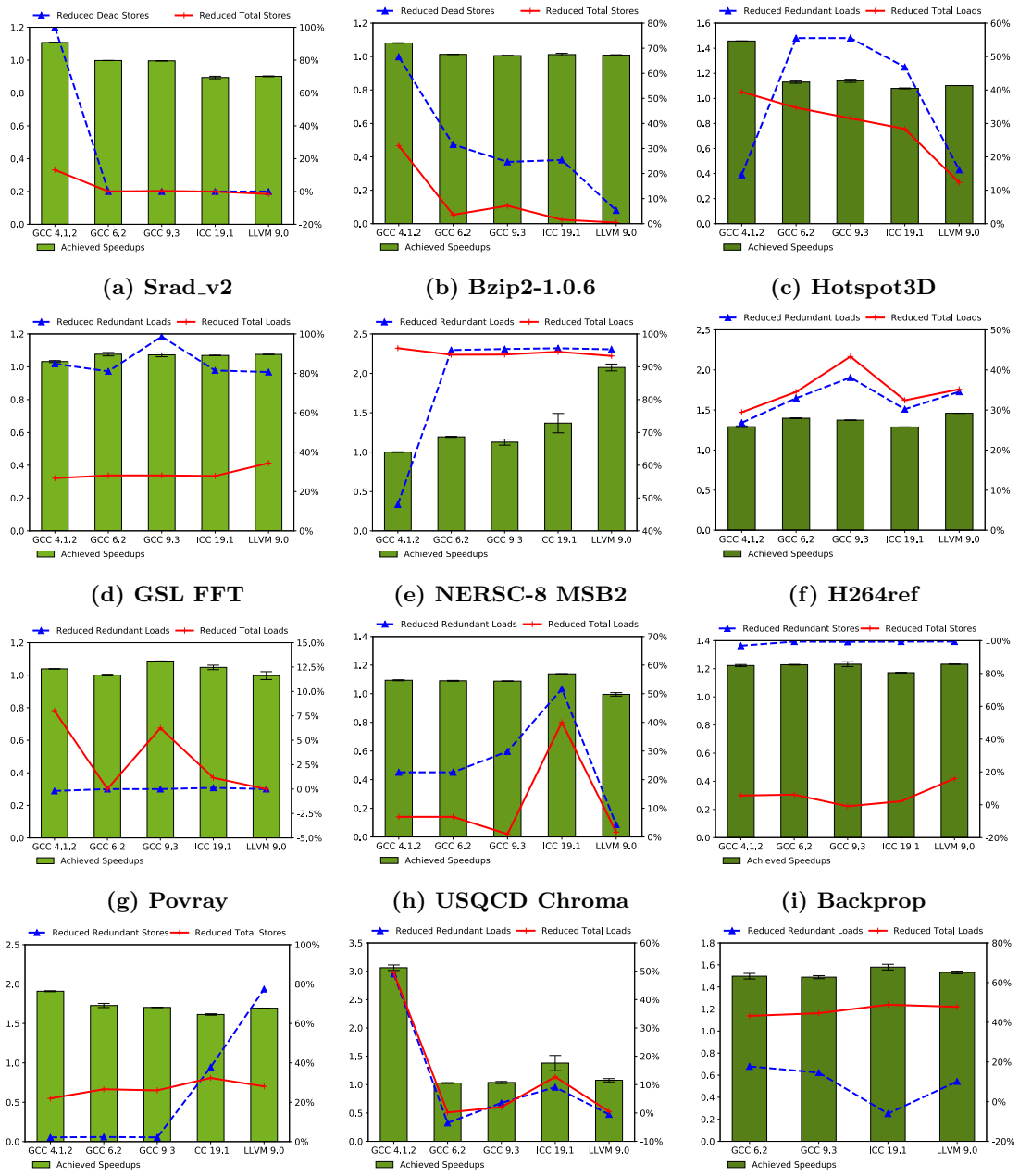
Figure 2.2: Experimental results for Hmmer. The left y-axis denotes speedups and the right y-axis denotes the percentage reduction in dead stores and total stores. The error bars denote the standard deviation across three executions.

It is worth noting that we do not compare the absolute execution time across compilers because different compilers apply different optimization techniques. Instead, we only compare the speedups and the percentage of reduction in loads or stores after our optimization on the inefficiencies described in Section 2.4. The speedups are averaged across three executions. We discuss the empirical results for each inefficiency category. Section 2.5.1 and 2.5.2 discuss the compiler-introduced inefficiencies and other sections describe the compiler-missed optimizations.

2.5.1 Inefficiencies due to Value Hoisting

2.5.1.1 Hmmer

Figure 2.2 shows the experimental results of Hmmer, including the optimization speedups, dead store reduction, and total store reduction. All the data are collected across five compilers. We describe our vertical and horizontal studies.



(a) Srad_v2 (b) Bzip2-1.0.6 (c) Hotspot3D

(d) GSL FFT (e) NERSC-8 MSB2 (f) H264ref

(g) Povray (h) USQCD Chroma (i) Backprop

(j) LavaMD (k) NERSC-8 MSB1 (l) Hoard

Figure 2.3: Experimental results for programs in CIBench. The left y-axis denotes speedups and the right y-axis denotes the percentage reduction of dead/redundant stores and total stores. The error bars denote the standard deviations across three executions.

Vertical study. All the `gcc` versions perform the value hoisting in Hmmer, which introduces dead store operations as `gcc 4.1.2`. These `gcc` compilers fail to reason about the aliases across the five arrays involved in the computation (see Listing 2.1) so all of them perform the conservative optimization as described in Section 2.4.1. Our optimization eliminates nearly all the dead store operations (99%) and reduces 50-64% total stores for all the `gcc` compilers, yielding 1.25-2.3× speedups.

Horizontal study. Our optimization on Hmmer receives speedups with both `gcc 9.3` and `llvm 9.0`. The nearly 100% elimination in dead stores and almost 70% of store reduction show the provenance of the speedups for `llvm 9.0`, which is similar to `gcc 6.2` and `gcc 9.3`. However, `icc 19.1` does not yield any speedup; instead, it slows the execution down by 53%. Statistical data show that our optimization reduces 32% dead stores and decreases the total stores by 4%. With further investigation, we find that our optimization by introducing the scalar variable in Listing 2.3 hurts the loop vectorization. Thus, our optimization is not generally applicable to all compilers. For `icc 19.1`, a better way that declares all the array involved in the computation with "restrict" keyword can both avoid producing most dead stores and generate better vector code, which yields a 20% speedup. It is worth noting that the "restrict" keyword eliminating inefficiencies only works for `icc`, not for `gcc` and `llvm`.

2.5.1.2 Srad_v2

Figures 2.3a shows the data of Srad_v2. Unlike Hmmer, the inefficiencies in Srad_v2 can be automatically optimized by some compilers.

Vertical study. For srad_v2, `gcc 6.2` and `gcc 9.3` do not introduce the inefficient code seen in `gcc 4.1.2`. From figure 2.3a, we can see that our optimization does not reduce any dead stores or total stores, and obtains no speedups. Further investigation shows that although srad_v2 has a similar code shape as Hmmer, it has only one array involved in

the computation (as shown in Listing 2.4). Recent `gcc` compilers are able to figure out the alias and avoid producing redundant operations.

Horizontal study. All the latest compilers can successfully resolve the aliasing issues for srad_v2, so our code optimization neither reduces dead stores nor reduces total stores, producing no speedups. Furthermore, like Hmmer, both `icc 19.1` and `llvm 9.0` have 10% slowdown with our optimized code, mainly due to the loss of vectorization.

2.5.2 Inefficiencies due to Instruction Scheduling

Vertical study. Both `gcc 6.2` and `gcc 9.3` do not aggressively schedule instructions in Bzip2 so they do not suffer from dead stores introduced by `gcc 4.1.2`. From Figure 2.3b we can see that `gcc 6.2` and `gcc 9.3` do not affect the total stores much, yielding no speedups with our optimization. It is worth noting that there are some reductions in dead stores after our optimization. As the absolute percentage of dead stores is small, it does not have much performance effect.

Horizontal study. All the latest compilers do not aggressively move instructions in Bzip2 so as to not introduce dead stores. Our optimization may introduce or eliminate some small amount of dead stores but with no effect on performance because the total amount of dead stores are small. As shown in Figure 2.3b, there is no performance impact for all three latest compilers.

2.5.3 Missing Scalar Replacement

Vertical study. As shown in Figure 2.3c, 2.3d, and 2.3e, the optimizations on Hotspot3D, GSL FFT, and MSB2 show speedups for all `gcc` versions, which means the evolution of `gcc` does not fully resolve the scalar replacement issues.

For Hotspot3D, the total loads reduced and speedups obtained from our optimization is shrinking with the evolution of `gcc` compilers. With the binary code analysis, we find

that the recent `gcc` versions adopt more aggressive loop unrolling strategies, which can reduce the redundant loads across iterations.

For GSL FFT with our optimization, all `gcc` versions eliminate most of the redundant loads that account for ∼30% total loads, yielding 3-8% speedups. It is because all the `gcc` versions do not resolve the aliases as described in Section 2.4.3.

For MSB2, all `gcc` versions do not automatically optimize the accesses to the static variable. Our optimization reduces almost 95% redundant loads and 96% total loads, yielding 1.2× and 1.1× speedups for `gcc 6.2` and `gcc 9.3`. It is worth noting that our optimization, although reduces redundant loads and total loads, yields a little speedup for `gcc 4.1.2`. The reason is that MSB suffers another inefficiency that is described in Section 2.4.4 as MSB1. That inefficiency involves arrays, which have a higher influence on performance. Both `gcc 6.2` and `gcc 9.3` do not suffer from MSB1 inefficiencies so they have speedups. Moreover, our optimization on an optimized MSB1 can yield a 17% speedup for `gcc 4.1.2`.

Horizontal study. Among all the compilers, `icc 19.1` most aggressively unrolls the loop and vectorizes its computation. Our optimization (scalar replacement) on Hotspot3D, although reduces some redundant load operations, it hurts the vectorization optimization. Thus, the optimization yields trivial speedup with `icc 19.1`. In contrast, `llvm 9.0` has similar behavior as `gcc 9.3`. For GSL FFT and MSB2, all the three compilers have similar behaviors: none of them eliminates the redundancies. Our optimization eliminates most of the redundancies and yields nontrivial speedups.

2.5.4 Missing Function Inlining

Vertical study. For Povray, all the `gcc` versions do not inline the target functions in H264ref. As shown in Figure 2.3f, newer `gcc` versions yield higher speedups; this is possibly because the new `gcc` versions optimize other parts of the program and the function inlining yields higher performance impacts.

For Chroma, all `gcc` compilers do not inline the problematic function. Our inlining optimization reduces both redundant loads and total load operations, yielding 1.09× speedups. For Povray, neither `gcc 4.1.2` nor `gcc 9.3` inlines the problematic function. Thus, our optimization achieves 4-9% speedups. However, `gcc 6.2` can automatically inline the function, which makes our optimization having no performance impacts.

Horizontal study. For Povray, `gcc 9.3` and `icc` do not inline the problematic function; our optimization reduces redundant load operations and yields 9% and 5% speedups for these two compilers. In contrast, `llvm` inlines the functions, and our hand optimizated code and compiler generated code have performance parity.

For H264ref, `icc` has similar behaviors as `gcc`, which misses the inlining opportunities. Our optimization with manual function inlining can remove many redundant operations and yield significant speedups. However, `llvm 9.0` has a different behavior. With LTO and PGO enabled, `llvm 9.0` is able to inline the indirect function call in H264ref but introduces a condition check in the loop body, resulting in only 8% speedup to the entire program. CIDetector still reports redundancies at the same place. Our manual optimization clones the loops and hoists the condition check out of the loops, resulting in an additional 46% speedup.

For Chroma, our manual optimization does not obtain any speedup when compiled with `llvm 9.0`, but incurs slightly slowdown. Moreover, there is almost no reduction in redundant loads and total load operations. Further investigation shows that `llvm` does not perform aggressive function inlining. Both `gcc 9.3` and `icc 19.1` inline all the callers of function `sranf`, while `llvm 9.0` has a few callers not inlined. Thus, `llvm 9.0`'s different inlining strategy results in different performance behavior.

2.5.5 Missing Constant Propagation

Vertical study. Backprop and LavaMD show similar behaviors across all the three `gcc` versions, which do not eliminate identity computation. Our manual optimization

yields significant speedups as shown in Figure 2.3i and 2.3j. For these two programs, our optimization either removes most of the redundant stores or many memory accesses. Moreover, optimizing the inefficiency of this category also removes redundant arithmetic computations that depend on the redundant memory loads or stores, resulting in more speedups.

For MSB1, the missing constant propagation that is not optimized by `gcc 4.1.2` is mostly optimized by `gcc 6.2` and `gcc 9.3`. Figure 2.3k shows that our optimization does not have an obvious impact on redundant loads, total load operations, and speedups.

Horizontal study. Like the vertical study, `gcc 9.3`, `icc 19.1`, and `llvm 9.0` do not eliminate the identity computations in Backprop and LavaMD; as a result, our optimization reduces many redundancies or total store operations and yields significant speedups.

For MSB1, by investigating the assembly code, we find `llvm 9.0` performs a similar optimization as `gcc 6.2` and `gcc 9.3`. Thus, our optimization yields a slight 1.08× speedup with `llvm 9.0`. In contrast, `icc 19.1` has a different behavior, which unrolls the loop with a factor of eight at line 8 in Listing 2.9. Then in each loop iteration, `icc` loads the value from `cache_buf[i]` into a register and assigns this register value to `cache_buf[i+1]`...`cache_buf[i+7]`. With this compiler optimization, the redundant loads can be reduced to 1/8. Our manual optimization can further eliminate the remaining redundancies and achieve a 38% speedup.

2.5.6 Missing Interprocedural Optimization

Hoard does not compile with `gcc 4.1.2` due to the lack of c++11 support, so for the vertical study, we only compare `gcc 6.2` and `gcc 9.3`. From Figure 2.3l, we can see that our manual optimization reduces redundant loads and yields ∼1.5× improvement in throughputs because both compilers fail to eliminate the redundant checks across function calls. For the horizontal study, all the three latest compilers do not eliminate redundant check operations automatically. Our manual optimization can eliminate nearly half total

loads and always yields significant throughput improvement.

Table 2.2: Optimization capabilities of different compilers.

Program Name	Hmmer	Srad_v2	Bzip2	LavaMD	Backprop	H264ref	Hotspot3D	Hoard	MSB1	MSB2	FFT	Povray	Chroma
gcc 9.3	N	S	S	N	N	N	P	N	S	N	N	N	N
icc 19.1	N	S	S	N	N	N	P	N	P	N	N	N	N
llvm 9.0	N	S	S	N	N	P	P	N	S	N	N	S	N

N: Not Optimized; P: Partially Optimized; S: Successfully Optimized.

2.6 Insights from the Study

The observation obtained from our experiments results in a number of insights regarding compiler-related inefficiencies. We discuss the insights that are useful to scientific programmers, compiler writers, and tool developers, respectively. Our discussion also answers questions raised in Section 2.1.

2.6.1 Insights for Scientific Programmers

This study offers four key insights for scientific programmers, which pertain to questions (4)-(6) in Section 2.1.

Inefficiencies exist in fully optimized binary codes. Modern compilers do not eliminate many inefficiencies. Even worse, inefficiencies can be introduced by compilers. Such inefficiencies can significantly degrade performance. These inefficiencies exist in a fully optimized binary code that is generated by mainstream compilers, including different versions of `gcc` compiler and the latest versions of `gcc`, `icc`, and `llvm`. Scientific programmers need to become aware of these compiler-related inefficiencies, which are only visible in the binary code. They can use tools such as [23, 190, 192] to regularly inspect the code quality as the software evolves and introduces complex constructs.

Finding inefficiency patterns in source code is difficult which makes tools necessary. Our studies show that it is difficult to find the inefficiency patterns in program

source code. Particularly, such patterns do not exist for compiler-introduced inefficiencies. Instead, one might be able to extract some patterns from the binary analysis, but pointers and aliases complicate the static analysis. Thus, a dynamic binary analysis tool like CIDetector is needed. Such a tool can glean rich information to guide source code revision in order to avoid compiler-introduced inefficiencies, or produce a code shape that enables important compiler optimizations.

Inefficiencies can be removed by curating source code. Our studies show that both compiler-introduced and compiler-missed inefficiencies optimizations can be removed by source code optimization. From Table 2.1, we can see that only a few lines of source code (2-62 with a median of 10) need to be changed for optimization.

Source code optimization does not always yield portable performance. Manual optimization may not yield portable performance for different compilers. For example, our optimization (i.e. scalar replacement) on Hmmer and Srad_v2 does not yield any speedup for `icc 19.1`, as shown in Section 2.4.3. Even though our optimization eliminates redundancies, the entire program undergoes nontrivial slowdowns when compiled with `icc 19.1`. As we discussed, such slowdowns are caused by the loss of vectorization which surpasses the benefit from redundancy elimination. A similar situation occurs for `llvm 9.0` on Chroma. Thus, manual optimization may hurt compiler optimization (e.g., vectorization in `icc 19.1` and inlining in `llvm 9.0`). Profiling is necessary to understand such a tradeoff between manual and compiler optimizations. Furthermore, one needs to curate code for the compiler they intend to use.

2.6.2 Insights for Compiler Writers

Our insights for compilers answer questions (1)-(3) in Section 2.1.

Insights from the vertical study. Generally speaking, `gcc 6.2` and `gcc 9.3` eliminate more inefficiencies compared to the older version `gcc 4.1.2` in the following three

aspects.

- `gcc` has improved the aliasing analysis in newer versions so that when fewer arrays are involved in the computation (e.g., srad_v2), they can avoid generating compiler-introduced inefficiencies.
- `gcc` avoids aggressive instruction scheduling in its newer versions (e.g., Bzip2).
- `gcc` has improved constant propagation optimization in its newer versions (e.g., MSB1).

Most inefficiencies shown in CIBench are not optimized by `gcc 6.2` and `gcc 9.3`. These inefficiencies include scalar replacement due to imprecise aliasing analysis (e.g., Hmmer, GSL FFT, MSB2), missing function inlining (e.g., H264ref, Chroma), and identity computation (e.g., LavaMD, Backprop). A future direction of `gcc` evolution can improve these inefficiencies.

Insights from the horizontal study. Our horizontal studies show that no compiler is better or worse than another production compiler in all cases. Table 2.2 summarizes all the programs optimized by different compilers or not.

- `icc 19.1` performs more aggressive loop unrolling that can reduce the redundant operations across loop iterations (e.g., LavaMD, MSB1).
- `llvm 9.0` has the most aggressive optimization with function inlining. For example, `llvm 9.0` optimizes Povray and partially optimizes H264ref. The inlining strategy in Chroma is also different from the other two compilers.
- `icc 19.1` is the only compiler that does not fully optimize MSB1 with constant propagation.

Thus, different compilers can borrow optimization techniques from each other in future releases. Moreover, most (69%) inefficiencies are not optimized by these recent compilers, so there is still large room for future compiler improvement.

2.6.3 Insights for Tool Developers

For our study, we can see the importance of a tool like CIDetector for analyzing the fully optimized binary code produced by different compilers. Given the optimization may not be portable, software engineers are suggested to frequently use tools upon committing code updates and changing development environments. Moreover, there are two possible directions for tool evolution. First, tools should evaluate the optimization tradeoff between dead/redundant operations with other inefficiencies. Second, tools should develop new metrics that quantify the speedups after optimization.

2.7 Related Work

There exist many approaches to analyzing inefficiencies. Some approaches [36, 42, 98, 76, 49, 48] analyze the code statically to identify redundancies. These static techniques yield many false positives and false negatives due to limitations such as imprecise alias analysis. To address the limitation, dynamic approaches [23, 190, 164, 191, 110, 120, 30, 88, 72, 43, 123, 117, 153, 192] pinpoint redundancies at runtime. Our tool CIDetector also uses dynamic analysis but distinguishes existing work in three aspects. First, unlike CIDetector, none of the existing tools can identify all the three kinds of inefficiencies in the fully optimized binary code generated by different modern compilers. Second, many of the existing tools pinpoint semantic-level redundancies, such as inappropriate data structures, suboptimal algorithms, and skewed inputs, not particularly focusing on the compiler-related inefficiencies. To the best of our knowledge, CIBench is the first data set for this purpose. Third, we perform the first study on the compiler vertical evolution and horizontal comparison and provide insights for different audiences.

Some prior studies identify performance bugs in real-world programs [83, 95, 150, 93, 68, 73] written in C, C++, Java, and JavaScript. They can extract some patterns in performance bugs and define rules to optimize them. Our approaches differ from them in two ways. First, we study the compiler-related performance bugs in binary code, which are

not systematically studied previously. Unlike existing approaches, the code repositories have few commits related to this kind of inefficiency. Thus, we develop CIDetector and CIBench to help the study. Second, no existing approaches study the inefficiencies produced by different compilers. Instead, they focus on software evolution.

The approach [100] perhaps is the most related work to ours. It evaluates the vectorization capabilities of different compilers on a set of benchmarks. Unlike their approach, we focus on compiler-related inefficiencies, which are orthogonal to vectorization.

2.8 Summary

In this chapter, we developed CIDetector and CIBench and leveraged them to perform the first study on compiler-introduced and compiler-missed optimizations in fully optimized binary codes. We studied five state-of-the-art compilers, including three `gcc` compilers of different versions, and the most recent `icc` and `llvm` releases. To the best of our knowledge, our work is the first systematic study of compiler-related inefficiencies. Our study offers several insights that are valuable for scientific programmers, compiler writers, and tool developers, including inspiring programmers of compiler limitations, showing that different compilers have different optimization strategies, and motivating the necessity of analysis tools. We expect the community to develop automatic optimizations to match or beat the performance of our hand optimization. CIDetector is integrated into CCTLib [22], available at `https://github.com/CCTLib/cctlib`; CIBench is available at `https://github.com/CCTLib/cibench`.

Chapter 3

Toward Efficient Interactions between Python and Native Libraries

3.1 Introduction

In recent years, Python has become the most prominent programming language for data modeling and library development, especially in the area of machine learning, thanks to its elegant design that offers high-level abstraction, and its powerful interoperability with native libraries that delivers heavy numeric computations. Decoupling data analysis and modeling logics from operation logics is the singular mechanism guiding the remarkable improvements in developers' productivity in the past decade. Python enables small teams to build sophisticated model [114] that were barely imaginable a few years ago, and enables large teams of modelers and numeric developers to seamlessly collaborate and develop highly influential frameworks such as Tensorflow [1] and Pytorch [125].

While high-level languages to articulate business logics and native libraries to deliver efficient computation is not a new paradigm, downstream developers have not always understood the details of native libraries, and have implemented algorithms that interacted

poorly with native codes. A well-known example of the *interaction inefficiency* problem occurs when developers, who fail to recognize that certain matrix operations can be vectorized, write significantly slower loop-based solutions. MATLAB and Mathematica can alleviate the problem since these languages usually are locked with a fixed set of native libraries over a long time, and developers can establish simple best practice guidelines to eliminate most interaction inefficiencies (MATLAB contains the command, "try to vectorize whenever possible").

In the Python ecosystem, native libraries and downstream application codes evolve rapidly so they can interact in numerous and unexpected ways. Therefore, building a list to exhaust all interaction inefficiencies becomes infeasible. We seek a solution that will automatically identify the blocks of Python code that lead to inefficient interactions, through closing the knowledge gap between Python and native code. Existing profiling tools cannot address this issue. Python profiles [58, 197, 61, 174, 143, 63, 129, 136, 172] cannot step in native code so they do not know execution details. Native profiling tools [141, 41, 118, 2, 23, 190, 163, 191] can identify hotspots, which offer insights into problematic code blocks. However, because these tools do not have knowledge about Python code's semantics, they cannot render detailed root cause and thus often make debugging remarkably challenging.

We propose PiePROF, the first lightweight, insightful profiler to pinpoint interaction inefficiencies in Python programs. PiePROF works for production Python software packages running in commodity CPU processors without modifying the software stacks. Its backbones algorithmic module is a recently proposed technique based on hardware performance monitoring units (PMUs) and debug registers to efficiently identify redundant memory accesses (hereafter, referred to as CL-algorithm[1] [191, 162]). CL-algorithm intelligently chooses a small collection of memory cells and uses hardware to track accesses to these cells at a fine granularity. For example, when the technique detects two consecutive writes of the same value to the same cell, it determines that the second write

[1] Chabbi-Liu Algorithm.

is unnecessary, and flags the responsible statement/function for further inspection. The developer can clearly see where a non-opt memory access occurs and why. The technique already shows its potential for eliminating inefficiencies in monolithic codebases that use one programming language.

PieProf leverages the CL-algorithm in a substantially more complex multi-languages environment, in which a dynamic and (predominantly) interpretation-based language Python is used to govern the semantics and native libraries compiled from C, C++, Fortran are used to execute high-performance computation. Doing so requires us to address three major challenges that crosscut Python and native code.

At the measurement front, we need to suppress false positives and avoid tracking irrelevant memory operations produced from Python interpreter and Python-native interactions. For example, memory accesses performed by Python interpreters may "bait" the CL-algorithm to waste resources (i.e., debug registers) on irrelevant variables such as reference counters. At the infrastructure front, we need to penetrate entire software stacks: it cannot see execution details (i.e, how memory is accessed) with only Python runtime information, or cannot understand program semantics with only native library knowledge. Our main task here is to compactly implement lock-free calling context trees that span both Python code and native libraries, and retain a large amount of information to effectively correlate redundant memory accesses with inefficient interactions. At the memory/safety front, we need to avoid unexpected behaviors and errors caused by Python runtime. For example, Python's garbage collection (GC) may reclaim memory that our tool is tracking. So delicate coordination between PieProf and Python interpreter is needed to avoid unexpected behaviors and errors.

We note that while most of the downstream applications we examined are machine learning related, PieProf is a generic tool that can be used in any codebase that requires Python-native library interactions.

Contributions. In this work, we make the following contributions.

- We are the first to thoroughly study the interaction inefficiencies between Python codes and native libraries. We categorize the interaction inefficiencies by their root causes.

- We design and implement PieProf, the first profiler to identify interaction inefficiencies and provide intuitive optimization guidance, by carefully stepping through Python runtimes and native binaries. PieProf works for production Python software packages in commodity CPU processors without modifying the software stacks.

- Following the guidance of PieProf, we examine a wide range of influential codebases and identify interaction inefficiencies in 17 real-world applications and optimize them for nontrivial speedups.

3.2 Background and Related Work

3.2.1 Python Runtime System

Python basics. Python is an interpreted language with dynamic features. When running a Python application, the interpreter translates Python source code into stack-based bytecode and executes it on the Python virtual machine (PVM), which varies implementations such as CPython [35], Jython [87], Intel Python [78] and PyPy [166]. This work focuses on CPython because it is *the reference implementation* [60], while the proposed techniques are generally applicable to other Python implementations as well. The CPython PVM maintains the execution call stack that consists of a chain of `PyFrame` objects known as function frames. Each `PyFrame` object includes the executing context of corresponding function call, such as local variables, last call instruction, source code file, and current executing code line, which can be leveraged by performance or debugging tools.

Python supports multi-threaded programming, where each Python thread has an individual call stack. Because of the global interpreter lock (GIL) [59], the concurrent execution of Python threads is emulated as regular switching threads by the interpreter,

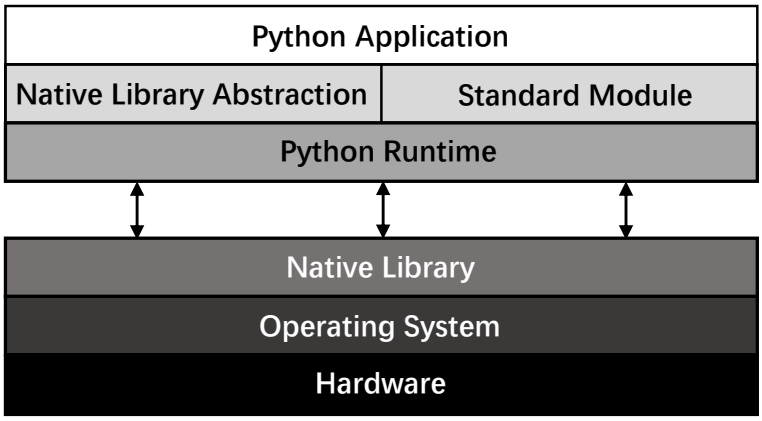


Figure 3.1: The typical stack of production Python software packages. Python applications usually rely on native libraries for high performance but introduce an abstraction across the boundary of Python runtime and native libraries.

i.e., for one interpreter instance, only one Python thread is allowed to execute at a time.

Interaction with native libraries. When heavy-lifting computation is needed, Python applications usually integrate native libraries written in C/C++/Fortran for computation kernels, as shown in Figure 3.1. Such libraries include Numpy [179, 69], Scikit-learn [127], Tensorflow [1], and PyTorch [125]. Therefore, modern software packages enjoy the benefit from the simplicity and flexibility of Python and native library performance. When the Python runtime calls a native function, it passes the `PyObject`[2] or its subclass objects to the native function. The Python runtime treats the native functions as blackboxes — the Python code is blocked from execution until the native function returns.

Figure 3.1 shows an abstraction across the boundary of Python runtime and native library, which logically splits the entire software stack. On the upper level, Python applications are disjoint from their execution behaviors because Python runtime (e.g., interpreter and GC) hides most of the execution details. On the lower level, the native libraries lose most program semantic information. This knowledge gap leads to interaction inefficiencies.

[2] `PyObject` is the super class of all objects in Python.

3.2.2 Existing Tools vs. PieProf

This section compares existing tools that analyze inefficiencies in Python and native codes to distinguish PieProf.

Python performance analysis tools. PyExZ3 [81], PySym [64], flake8 [37], and Frosted [169] analyze Python source code and employ multiple heuristics to identify code issues statically [67]. XLA [167] and TVM [26] apply compiler techniques to optimize deep learning applications. Harp [196] detects inefficiencies in Tensorflow and PyTorch applications based on computation graphs. All of these approaches, however, ignore Python dynamic behavior, omitting optimization opportunities.

Dynamic profilers are a complementary approach. cProfile [58] measures Python code execution, which provides the frequency/time executions of specific code regions. Guppy [197] employs object-centric profiling, which associates metrics such as allocation frequency, allocation size, and cumulative memory consumption with each Python object. PyInstrument [143] and Austin [172] capture Python call stack frames periodically to identify executing/memory hotspots in Python code. PySpy [61] is able to attach to a Python process and pinpoint function hotspots in real time. Unlike PieProf, these profilers mainly focus on Python codes, with no insights into the native libraries.

Closely related to PieProf, Scalene [14] separately attributes Python/native executing time and memory consumption. However, it does not distinguish useful/wasteful resources usage as PieProf does.

Native performance analysis tools. While there are many native profiling tools [141, 41, 2], from which the most related to Python that can identify performance inefficiencies are Toddler [118] that identifies redundant memory loads across loop iterations, and LDoctor [154] that reduces Toddler's overhead by applying dynamic sampling and static analysis. DeadSpy [23], RedSpy [190], and LoadSpy [163] analyze dynamic instructions in the entire program execution to detect useless computations or data movements. Un-

fortunately, all of them use heavyweight binary instrumentation, which results in high measurement overhead, and they do not work directly on Python programs.

3.2.3 Performance Monitoring Units and Hardware Debug Registers

Hardware performance monitoring units (PMUs) are widely equipped on the modern x86 CPU architectures. Software can use PMUs to count various hardware events like CPU cycles, cache misses, et cetera. Beside the counting mode that counts the total number of events, PMUs can be configured in sampling, which periodically sample a hardware event and record event's detailed information. PMUs trigger an overflow interrupt when the sample number reaches a threshold. The profiler runtime captures interrupts as signals and collects samples with their executing contexts.

For memory-related hardware events such as memory load and store, Precise Event-Based Sampling (PEBS) [38] in Intel processors provides the effective address and the precise instruction pointer for each sample. Instruction-Based Sampling (IBS) [50] in the AMD processors and Marked Events (MRK) [157] in PowerPC support similar functionalities.

Hardware debug registers [85, 102] trap the CPU execution when the program counter (PC) reaches an address (breakpoint) or an instruction accesses a designated address (watchpoint). One can configure the trap conditions with different accessing addresses, widths and types. The number of hardware debug registers is limited (e.g., the modern x86 processor has four debug registers).

3.3 Interaction Inefficiency Characterization

This section provides a high-level preview of the key findings from applying PieProf to an extensive collection of high-profile Python libraries at Github. We specifically categorize the interaction inefficiencies according to the root causes and summarize the common patterns, which serve three purposes: *(i)* this is the first characterization of interaction

```
def train(self, trainData, maxEpochs, learnRate):
    ...
    for j in range(self.nh):
        delta = -1.0 * learnRate * ihGrads[i,j]
        self.ihWeights[i, j] += delta
    ...
```

Listing 3.1: Interaction inefficiencies in IrisData due to the iteration on Numpy arrays within a loop.

inefficiencies based on large scale studies, thus rendering a more complete landscape of potential code quality issues that exist in Python codebase for ML and beyond; *(ii)* we see a diverse set of inefficiencies hiding deep in Python-native library interaction, which justifies using heavy machineries/profiling tools to automatically identify them; and *(iii)* these concrete examples explain the common patterns we use to drive the PieProf's design.

3.3.1 Interaction Inefficiency Categorization

We categorize interaction inefficiencies into five groups. For each category, we give a real example, analyze the root causes, and provide a fix.

Slice underutilization. Listing 3.1 is an example code from IrisData [161], a back-propagation algorithm implementation on Iris Dataset [56]. A loop iterates two multidimensional arrays `ihGrads` and `ihWeights` with indices `i` and `j` for computation. Because Python arrays are supported by native libraries such as Numpy and PyTorch/TensorFlow, indexing operations (i.e., `[]`) in a loop trigger native function calls that repeat boundary and type checks [119].

The so-called vectorization/slicing eliminates repeated "housework" and (usually) enables the underlying BLAS [17] library to perform multi-core computation. Listing 3.2 shows a simple fix in a 2× speedup for the entire program execution.

Repeated native function calls with the same arguments. Functions from native libraries typically have no side effects, so applying the same arguments to a native function

```
1 def train(self, trainData, maxEpochs, learnRate):
2     ...
3     self.ihWeights[i, 0:self.nh] += -1.0 * learnRate * ihGrads[i, 0:self.nh]
4     ...
```

Listing 3.2: Optimized IrisData code with slice notation.

```
1 def rotate(self, theta):
2     a = np.cos(theta)
3     b = np.sin(theta)
4     rotate_mtx = np.array([[a, -b, 0.0], [b, a, 0.0], [0.0, 0.0, 1.0]], float)
5     self._mtx = np.dot(rotate_mtx, self._mtx)
6     ...
```

Listing 3.3: Interaction inefficiencies in Matplotlib due to the same input `theta`.

results in the same return value, which introduces redundant computations. Listing 3.3 shows a code from Matplotlib [77], a comprehensive library for visualization and image manipulation. This code rotates an image and is often invoked in training neural nets for images.

The argument `theta` for the `rotate` function (rotate angle) is usually the same across consecutive invocations from deep learning training algorithms because they rotate images in the same batch in the same way. Here, `Pyobject`s returned from native functions `np.cos()`, `np.sin()` and `np.array()` in lines 2-4 have the same values across images that share the same input `theta`.

This can be fixed by either a simple caching trick [44, 113], or refactoring the `rotate` funcion so that it can take a batch of images. We gain a 2.8× speedup after the fix.

Inefficient algorithms. Listing 3.4 is an example of algorithmic inefficiencies from Scikit-learn, a widely used machine learning package. The code works on `X`, a two-dimensional Numpy array. It calls the native function `swap` from the BLAS library to exchange two adjacent vectors. In each iteration, `swap` returns two `PyObject`s and Python runtime assigns these two `PyObject`s to `X.T[i]` and `X.T[i+1]`, respectively. The loop uses `swap` to move the first element in the range to the end position. Inefficiencies occur because it requires multiple iterations to move `X.T[i]` to the final location.

Instead of using `swap`, we directly move each element to the target location. We apply

```
def lars_path(X, y, Xy=None, ...):
    ...
    for i in range(ii, n_active):
        X.T[i], X.T[i + 1] = swap(X.T[i], X.T[i + 1])
        indices[i], indices[i + 1] = indices[i + 1], indices[i]
    ...
```

Listing 3.4: Interaction inefficiencies in Scikit-learn due to the inefficient algorithm.

```
def CEC_4(solution=None, problem_size=None, shift=0):
    ...
    for i in range(dim - 1):
        res += 100 * np.square(x[i]**2-x[i+1]) + np.square(x[i]-1)
    ...
```

Listing 3.5: Interaction inefficiencies in Metaheuristic [115, 116] due to the API misuse in native Libraries.

a similar optimization to the `indices` array as well. Our improvement yields a 6.1× speedup to the `lars_path` function.

API misuse in native libraries. Listing 3.5 is an example of API misuse from Meta-heuristic [115, 116], which implements the-state-of-the-art meta-heuristic algorithms. The code accumulates the computation results to `res`. Since the computation is based on Numpy arrays, the accumulation operation triggers one native function call in each iteration, resulting in many inefficiencies.

In Listing 3.6 shows our fix (i.e., use the efficient `sum` API from Numpy) which avoids most of the native function invocations by directly operating on the Numpy arrays. This optimization removes most of interaction inefficiencies, and yields a 1.9× speedup to the entire program.

Loop-invariant computation. Listing 3.7 is a code snippet from Deep Dictionary Learning [99], which seeks multiple dictionaries at different image scales to capture complementary coherent characteristics implemented with TensorFlow. Lines 1-3 indicate the computation inputs `A`, `D`, and `X`. Lines 4-5 define the main computation. Lines 6-7 execute the computation with the actual parameters `D_` and `X_`. The following pseudo-code shows the implementation:

for $i \leftarrow 1$ to $Iter$ **do**

$$A = D(X - D^T A)$$

where `D` and `X` are loop invariants. If we expand the computation, DX and DD^T can be computed outside the loop and reused among iterations, shown as pseudo-code:

$t_1 = DX$

$t_2 = DD^T$

for $i \leftarrow 1$ to $Iter$ **do**

$$A = t_1 - t_2 A$$

This optimization yields a 3× speedup to the entire program [196].

3.3.2 Common Patterns in Interaction Inefficiencies

We are now ready to explain the common patterns in code that exhibits interaction efficiencies, which we use to drive the design of PieProf. Specifically, we find that almost all interaction inefficiencies involve *(i)* repeatedly reading the same `PyObject`s of the same values, and *(ii)* repeatedly returning `PyObject`s of the same values.

Both observations require developing a tool to identify redundant `PyObject`s, which is difficult and costly because it requires heavyweight Python instrumentation and modification to Python runtime. Further analysis, however, finds that `PyObject` redundancies reveal the following two low-level patterns during the execution from the hardware perspective.

Table 3.1: Redundant loads and stores detect different categories of interaction inefficiencies.

Inefficiency Pattern	Inefficiency Category
Redundant Loads	Slice underutilization
	Inefficient algorithms
	API misuse in native libraries
Redundant Stores	Loop-invariant computation
	Repeated native function calls with same arguments
	Inefficient algorithms
	API misuse in native libraries

```
def CEC_4(solution=None, problem_size=None, shift=0):
    ...
    res += np.sum(100 * np.square(x[0:dim-1]**2 -  x[1:dim]) + np.square(x[0:dim
        -1] - 1))
    ...
```

Listing 3.6: Optimized Metaheuritics code for Listing 3.5, with appropriate native library API.

```
A = tf.Variable(tf.zeros(shape=[N, N]), dtype=tf.float32)
D = tf.placeholder(shape=[N, N], dtype=tf.float32)
X = tf.placeholder(shape=[N, N], dtype=tf.float32)
R = tf.matmul(D, tf.subtract(X, tf.matmul(tf.transpose(D), A)))
L = tf.assign (A, R)
for i in range(Iter):
    result = sess.run(L, feed_dict={D: D_, X: X_})
```

Listing 3.7: Interaction inefficiencies in Deep Dictionary Learning [99] due to loop-invariant computation.

- *Redundant loads:* If two adjacent native function calls read the same value from the same memory location, the second native function call triggers a redundant (memory) load. Repeatedly reading `PyObject` of the same value result in redundant loads.
- *Redundant stores:* If two adjacent native function calls write the same value to the same memory location, the second native function call triggers a redundant (memory) store. Repeatedly returning `PyObject` of the same value result in redundant stores.

We use the redundant loads and stores to serve as indicators of interaction inefficiencies. Table 3.1 shows different categories of interaction inefficiencies, which show up as redundant loads or stores. Section 4 describes how we use the indicators.

3.4 Design and Implementation

3.4.1 Overview

See Figure 3.2. Recall that the CL-algorithm controls PMUs and debug registers to report redundant member accesses of a process. PiePROF interact with Python runtime, native libraries, and the CL-algorithm through three major components: *(i) Safeguard and sandbox.* A thin sandbox is built around Python interpreter and native libraries,

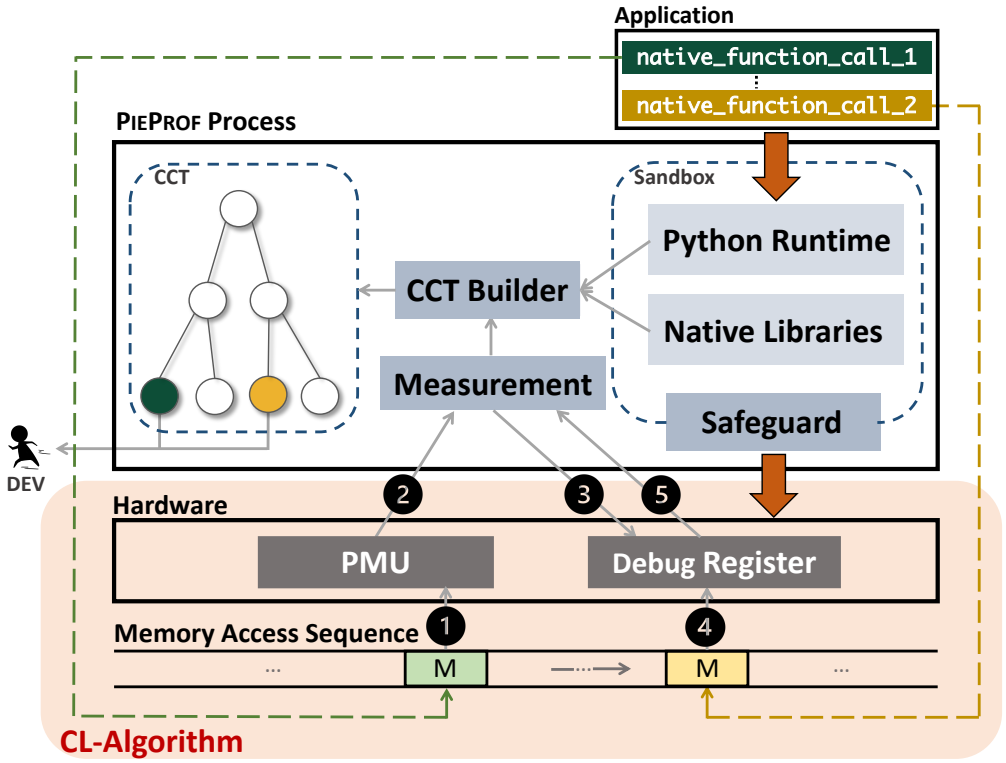


Figure 3.2: Overview of PieProf's workflow.

and a safeguard is implemented inside the sandbox to moderate communication between Python runtime and the CL-algorithm. *(ii) Measurement.* Upon receiving an event from the CL-algorithm, the measurement component determines whether to notify CCT (calling context tree) builder to update the CCT, and *(iii) CCT Builder.* Upon receiving an update from the measurement component, CCT builder examines Python runtime and native call stacks to update CCT. When an interaction inefficiency is detected, it will report to the end user (developer).

The measurement component helps to suppress false positive and avoid tracking irrelevant variables (e.g., reference counters), the CCT builder continuously updates the lock-free CCT, and Safeguard/sandbox ensures that the Python application can be executed without unexpected errors.

We next discuss each component in detail.

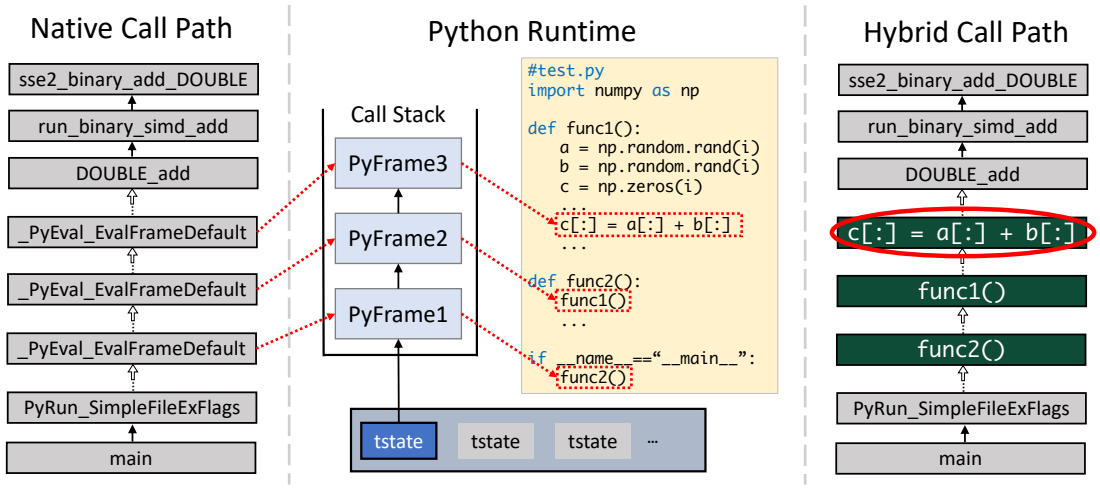


Figure 3.3: Constructing a hybrid call path across Python runtime and native libraries. White arrows in call paths denote a series of elided call frames in PVM. The red circle in the hybrid call path shows the boundary of Python and native frames, where interaction inefficiencies occur.

3.4.2 Measurement

CL-algorithm. CL-algorithm uses PMUs and debug registers to identify redundant loads and stores in an instruction stream. It implements a conceptually simple and elegant process: a sequence $a_1, a_2, \ldots, a_m$ memory access instructions arrive at the CL-algorithm in a streaming fashion. Here, a_i refers to the address of the memory access for the i-th instruction. Upon seeing a new memory access instruction a_i (step 1, i.e ❶ in Figure 3.2), the CL-algorithm uses PMUs to probabilistically determine whether it needs to be tracked (step 2), and if so, store the address in a debug register (step 3). If the debug registers are all used, a random one will be freed up. When a subsequent access to a_i (or any addresses tracked by debug registers) occurs (step 4), the debug register will trigger an interrupt so that the CL-algorithm can determine whether the access is redundant (step 5), by using the rules outlined in Section 3.3.2. Since the number of debug registers is usually limited, the CL-algorithm uses a reservoir sampling [186] technique to ensure that each instruction (and its associated memory accesses) has a uniform probability of being sampled.

Improving measurement efficiencies. First, PMUs sample instructions at the hardware level so it cannot distinguish memory accesses from the Python interpreter from those from the Python applications. In practice, a large fraction of memory access sequences are related to updating reference counters for Python objects. Therefore, most debug registers will be used to track reference counters if we bluntly use the CL-algorithm, and substantially reduces the chances of identifying memory access redundancies. Second, it needs to ignore redundant memory accesses occurring within the same native function call, or within a code region of PiePROF because they are not related to interaction inefficiencies. Note that tracking redundant memory accesses within the same native function call is worse than merely producing false positives because it can bury true instances. For example, two write instructions w_1 and w_2 of the same value are performed on the same memory from function F_a, and later function F_b performs a third write instruction w_3 of the same value on the same location. If we track redundant accesses within the same function, the CL-algorithm says it has found a redundant pair $\langle w_1, w_2 \rangle$, evicts w_1 from the debug register. and never detects the redundant pair $\langle w_1, w_3 \rangle$ caused by the real interaction inefficiencies.

PiePROF performs instruction-based filter to drop a sample if *(i)* its instruction pointer falls in the code region unrelated to native function calls (e.g., that of PiePROF), *(ii)* its memory access address belongs to "junky" range, such as the head of `PyObject` that contains the reference number. In addition, when the CL-algorithm delivers a redundant memory access pair to PiePROF, it checks the Python runtime states and drops the sample when these two memory accesses occur inside a same state (corresponding to within the same native function call).

3.4.3 Calling Context Trees Builder

This section first explains the construction of call paths, and then explains how they can be used to construct signal-free calling context trees (CCTs).

Hybrid call path. PiePROF uses libunwind [147] to unwind the native call path of a Python process to obtain a chain of procedure frames on the call stack. See the chain of "Native Call Path" on the left in Figure 3.3. Here, call stack unwinding is not directly applicable to Python code because of the abstraction introduced by PVM. The frames on the stack are from PVM, not Python codes. For example, the bottom `_PyEval_EvalFrameDefault`[3] shows up in "Native Call Path", but we need the call to correspond to `func2()` in Python code (connected through `PyFrame1`). Thus, PiePROF needs to inspect the dynamic runtime to map native calls with Python calls on the fly.

1. Mapping `PyFrame` *to Python calls.* First, we observe that each Python thread maintains its call stacks in a thread local object `PyThreadState` (i.e., `tstates` in Figure 3.3). To obtain Python's calling context, PiePROF first calls `GetThisThreadState()`[4] to get the `PyThreadState` object of the current thread. Second PiePROF obtains the bottom `PyFrame` object (corresponding to the most recently called function) in the PVM call stack from the `PyThreadState` object. All `PyFrame` objects in the PVM call stack are organized as a singly linked list so we may obtain the entire call stack by traversing from the bottom `PyFrame`. Each `PyFrame` object contains rich information about the current Python frame, such as source code files and line numbers that PiePROF can use to correlate a `PyFrame` to a Python method. In Figure 3.3, `PyFrame1`, `PyFrame2`, and `PyFrame3` are for Python methods `main`, `func2`, and `func1`, respectively.

2. Extracting `PyFrame`*'s from Native Call Path.* Each Python function call leaves a footprint of `_PyEval_EvalFrameDefault` in the native call stack so we need only examine `_PyEval_EvalFrameDefault`. Each `_PyEval_EvalFrameDefault` maps to a unique `PyFrame` in the call stack of the active thread in Python Runtime. In addition, the ordering preserves, e.g., the third `_PyEval_EvalFrameDefault` in "Native Call Path" corresponds to the third `PyFrame` in Python's call stack. Therefor use standard Python interpreter APIs to obtain the `PyFrame`'s and map them back to nodes in the native call path.

[3] `_PyEval_EvalFrameDefault` is a frame (i.e., a function pointer) in the native call stack in runtime that corresponds to invocation of a function or a line of code in Python.

[4] `GetThisThreadState()` is a PVM API to retrieve an object that contains the state of current thread.

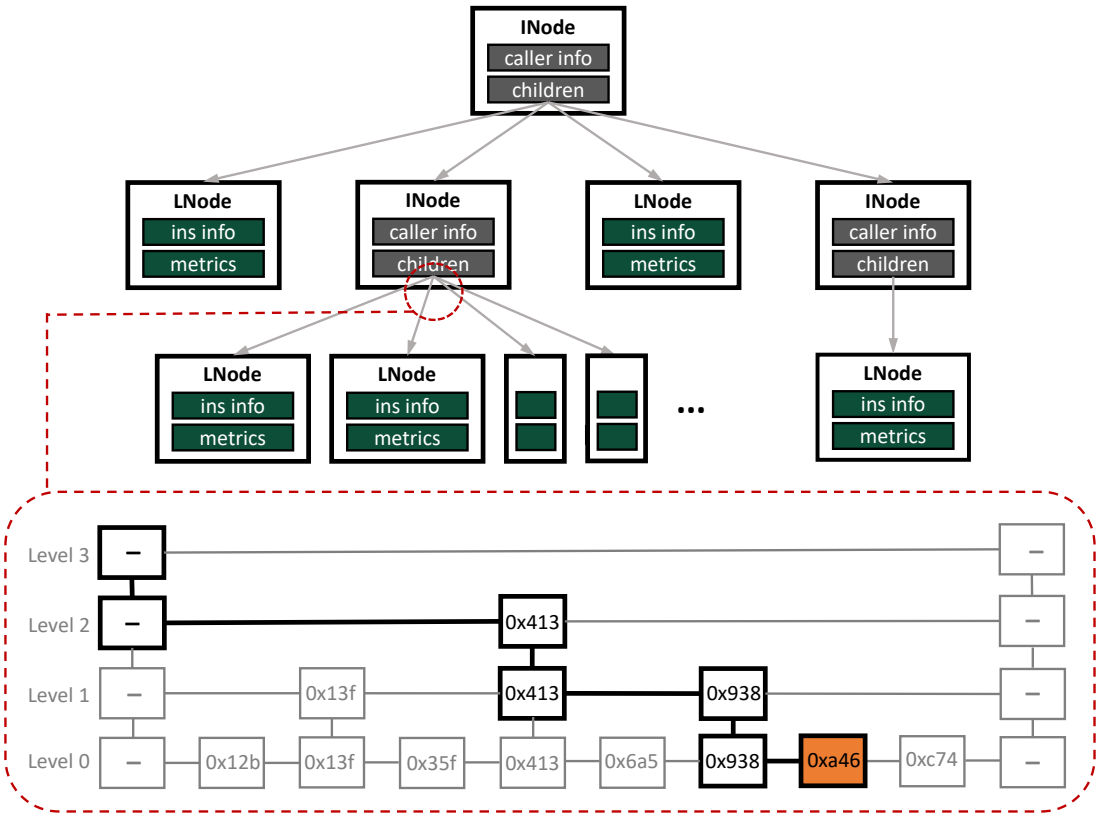


Figure 3.4: A calling context tree constructed by PieProf. Each parent node applies skip-list to organize children. `INode` denotes an internal node and `LNode` denotes a leaf node. Red box shows searching `0xa46` in the example skip-list.

CCT from call paths. PiePROF applies a compact CCT [7, 6] to represent the profile. Figure 3.4 shows the structure of a CCT produced by PiePROF. The internal nodes represent native or Python function calls, and the leaf nodes represents the sampled memory loads or stores. Logically, each path from a leaf node to the root represents a unique call path.

As mentioned, Python is a dynamic typing language, and uses meta-data to represent calling context (e.g., the function and file names in string form); therefore, its call stacks are usually substantially larger (in space) than those in static languages. One solution is to build a dictionary to map strings to integer ids but the solution must be signal-free because it needs to interact with the CL-algorithm and PMUs, which is prohibitively

complex.

Our crucial observation is that function calls in different threads near the root of a tree usually repeat so unlike solutions appeared in [23, 190, 163, 22, 162], which produce a CCT for each thread/process, PiePROF constructs a single CCT for the entire program execution. In this way, the same function call appearing in different threads is compressed into one node and space complexity is reduced. PiePROF also implements a lock-free/signal-safe skip-list [133] to maintain CCT's edges for fast and thread-safe operations. In theory, Skip-list's lookup, insert, and delete operations have $O(\log n)$ time complexity. In practice, Skip-list with more layers has higher performance but higher memory overhead. In a CCT, the nodes closer to the root are accessed more frequently. PiePROF, however,proportionally adjusts the number of layers in the skip-lists at different levels in a CCT to optimize the performance and overhead tradeoffs. It uses more layers to represent the adjacency lists of nodes that are close to the root, and fewer layers to represent those that are close to the leaves.

3.4.4 Safeguard

PiePROF uses two mechanisms to avoid unexpected errors in Python runtime. It will hibernate if it enters a block of code, interrupting which will cause state corruption in PVM, and will block certain activities from GC if the activities can cause memory issues.

Hibernation at function-level. Upon seeing an event (e.g., an instruction is sampled or a redundant memory access is detected), the PMUs or debug registers use interrupt signals to interact with PiePROF, which will pause Python's runtime. Error could happen if Python run time is performing certain specific tasks when an interrupt exception is produced. For example, if it is executing memory management APIs, memory error (e.g., segmentation fault) could happen, and if Python is loading native library, deadlock could happen.

PiePROF maintains a list of functions, inside which PiePROF needs to be temporarily

Table 3.2: Overview of performance improvement guided by PieProf. *AS* **denotes application-level speedup,** *FS* **denotes function-level speedup,** *L* **refers to redundant loads and** *S* **refers to redundant stores.**

Program Information			Inefficiency		Optimization	
Applications	Library	Problem Code	Category	Pattern	AS	FS
Ta [124]	Ta	volatily.py(45)/trend.py(536, 549, 557, 571, 579)	Slice underutilization	*L*	1.1×	16.6×
NumPyCNN [62]	Numpy [69, 179]	numpycnn.py(161)	Loop-invariant computation	*S*	1.8×	2.04×
Census_main	NumpyWDL [159]	ftrl.py(60)	Loop-invariant computation	*S*	1.03×	1.1×
Lasso	Scikit-learn [127]	least_angle.py(456, 458)	Inefficient algorithms	*S*	1.2×	6.1×
IrisData [161]	Numpy	nn_backprop.py(222, 228, 247, 256, 263, 271, 278)	Slice underutilization & API misuse	*L*	2×	2.02×
Network	Neural-network-from-scratch	network.py(103-115)	Repeated NFC	*L*	1.03×	1.05×
Cnn-from-scratch [195]	Numpy	conv.py(62)	Slice underutilization	*L*	2.5×	3.9×
Metaheuristics [115, 116]	Numpy	FunctionUtil.py(374)	API misuse	*L*	1.4×	1.9×
		FunctionUtil.py(270)	Slice underutilization	*L*	6.3×	27.3×
		FunctionUtil.py(309, 375)	Loop-invariant computation	*S*	1.04×	1.4×
		FunctionUtil.py(437)	Repeated NFC	*L*	1.02×	1.1×
		EPO.py(40)	Loop-invariant computation	*S*	1.1×	1.1×
LinearRegression [91]	LinearRegression	LinearRegression.py(49, 50)	Repeated NFC	*L*	1.4×	1.5×
Pytorch-examples [84]	PyTorch [125]	adam.py:loop(66)	Loop-invariant computation	*L*	1.02×	1.07×
Cholesky [196]	PyTorch	cholesky.py(76)	Slice underutilization	*L*	3.2×	3.9×
GGNN.pytorch [29]	PyTorch	model.py(122, 125)	Loop-invariant computation	*S*	1.03×	1.07×
Network-sliming [94]	Torchvision [137]	functional.py(164)	Slice underutilization	*L*	1.1×	1.7×
Pytorch-sliming [94]					1.04×	1.7×
Fourier-Transform [89]	Matplotlib [77]	transforms.py(1973)	Repeated NFC	*S*	1.02×	2.8×
Jax [19]					1.04×	2.8×
Autograd [66]					1.05×	2.8×

turned off (i.e., in hibernation mode). To do so, PieProf maintains a block list of function, and implements wrappers for each function in the list. Calls to these functions are redirected to the wrapper. The wrapper turns off PieProf, executes the original function, and turns on PieProf again.

Dropping events vs. hibernation. We sometimes drop an event when it is unwanted (Section 3.4.2). Complex logic can be wired to drop an event at the cost of increased overhead. Here, hibernating PieProf is preferred to reduce overhead because no event needs to be kept for a whole block of code.

Blocking garbage collector. When Python GC attempts to deallocate the memory that debug registers are tracking, errors could occur. Here, we uses a simple trick to defer garbage collection activities: when PieProf monitors memory addresses and it is within a `PyObject`, it increases the corresponding `PyObject`'s reference, and decreases the reference once the address is evicted. This ensures that memories being tracked will not be deallocated. Converting addresses to `PyObject`'s is done through progressively heavier mechanisms. First, `PyObject`'s exist only in a certain range of the memory so we can easily filter out addresses that do not correspond to `PyObject` (which will not be deallocated

by GC). Second, we can attempt to perform a dynamic casting on the address and will succeed if that corresponds to the start of an `PytObject`. This handles most of the cases. Finally, we can perform a full search in the allocator if we still cannot determine whether the address is within a `PyObject`.

3.5 Evaluation

This section studies the effectiveness of PiePROF (e.g., whether it can indeed identify interaction inefficiencies) and its overheads.

We evaluate PiePROF on a 14-core Intel Xeon E7-4830 v4 machine clocked at 2GHz running Linux 3.10. The machine is equipped with 256 GB of memory and four debug registers. PiePROF is compiled with `GCC 6.2.0 -O3`, and CPython (version 3.6) is built with `--enable-shared` flag. PiePROF subscribes hardware event `MEM_UOPS_RETIRED_ALL_STORES` for redundant stores detection and `MEM_UOPS_RETIRED_ALL_LOADS` for redundant loads detection, respectively.

3.5.1 Effectiveness

This section assesses the effectiveness of PiePROF, and the breadth of the interaction inefficiencies problem among influential Python packages. The lack of a public benchmark creates two inter-related challenges: *(i)* determining the codebases to examine inevitably involves human intervention, and *(ii)* most codebases provide a small number of "hello world" examples, which have limited test coverage.

We aim to include all "reasonably important" open-source projects and use only provided sample code for testing. While using only sample code makes inefficiency detection more difficult, this helps us to treat all libraries as uniformly as possible. For each of Numpy, Scikit-learn, and Pytorch, we find all projects in Github that import the library, and sort them by popularity, which gives us three lists of project candidates. Our stopping rule for each list differs and involves human judgement because we find that the popular-

ity of a project may not always reflect its importance (e.g., specialized libraries could be influential, but generally have smaller user bases, and are less popular in Github's rating system). For example, Metaheuristics is important and included in our experiment but it received only 91 ratings at the time we performed evaluation. At the end, we evaluated more than 70 read-world applications, among which there are more projects that import Numpy than the other two libraries.

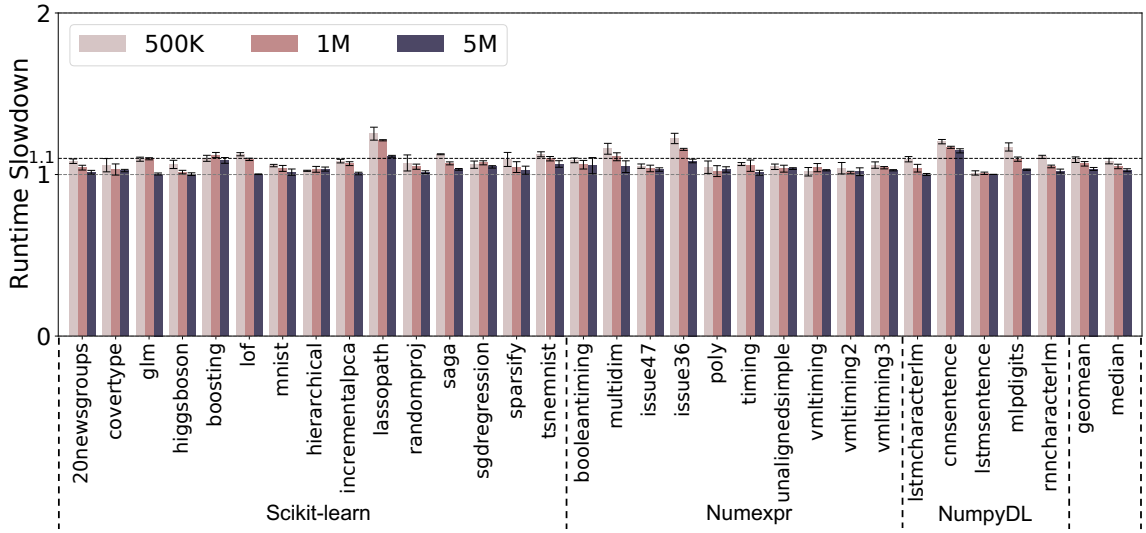
Indentifying a total of 19 inefficiencies is quite surprising because these projects are mostly written by professionals, and the sample codes usually have quite low codebase coverage, and are usually "happy paths" that are highly optimized. The fact that we identify 18 new performance bugs as reported in Table 2, indicates that interaction inefficiencies are quite widespreaded.

Table 3.2 reports that the optimizations following PiePROF's optimization guidance lead to 1.02× to 6.3× application-level speedup (AS), and 1.05× to 27.3× function-level speedup (FS), respectively. According to Amdahl's law, AS approaches FS as the function increasingly dominates the overall execution time. For the five inefficiency categories we define in Section 3.3.1 and which are common in real applications, PiePROF's superior redundant loads/stores detection proves its effectiveness.

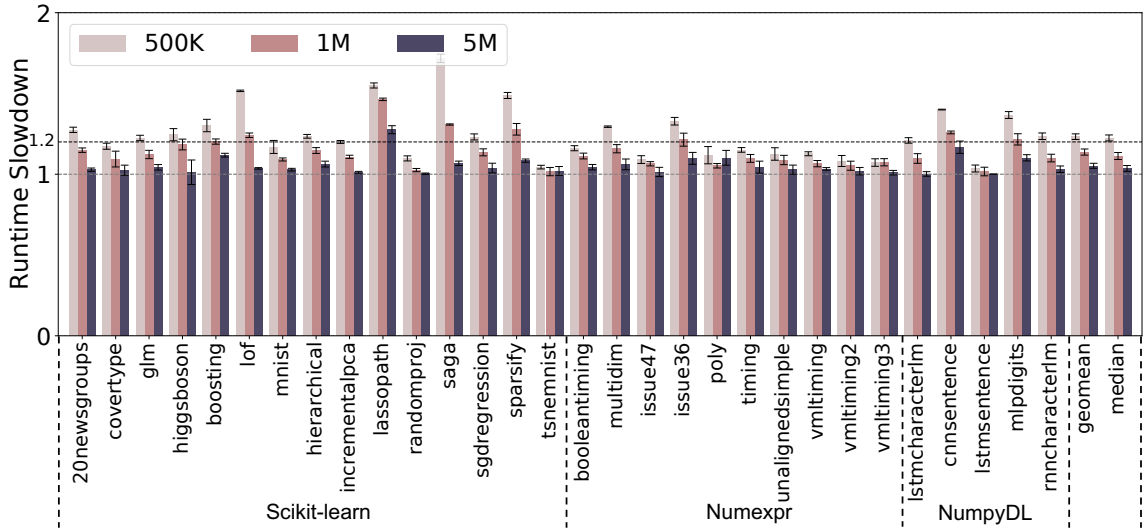
3.5.2 Overhead

This section reports the runtime slowdown and memory bloating caused by PiePROF. We measure runtime slowdown by the ratio of program execution time with PiePROF enabled over its vanilla execution time. Memory bloating shares the same measuring method but with the peak memory usage.

Since Python does not have standard benchmarks, we evaluate the overhead of PiePROF on three popular Python applications — Scikit-learn, Numexpr [134], and NumpyDL [187] which contain benchmark programs from scientific computing, numerical expression and deep learning domains. We report only the first half of the Scikit-



(a) Redundant Stores Detection

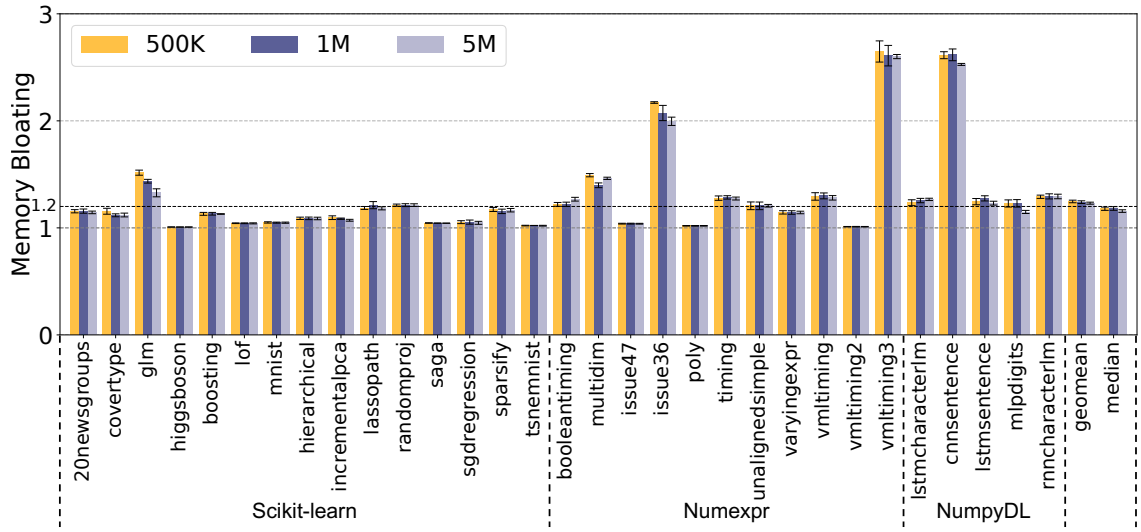


(b) Redundant Loads Detection

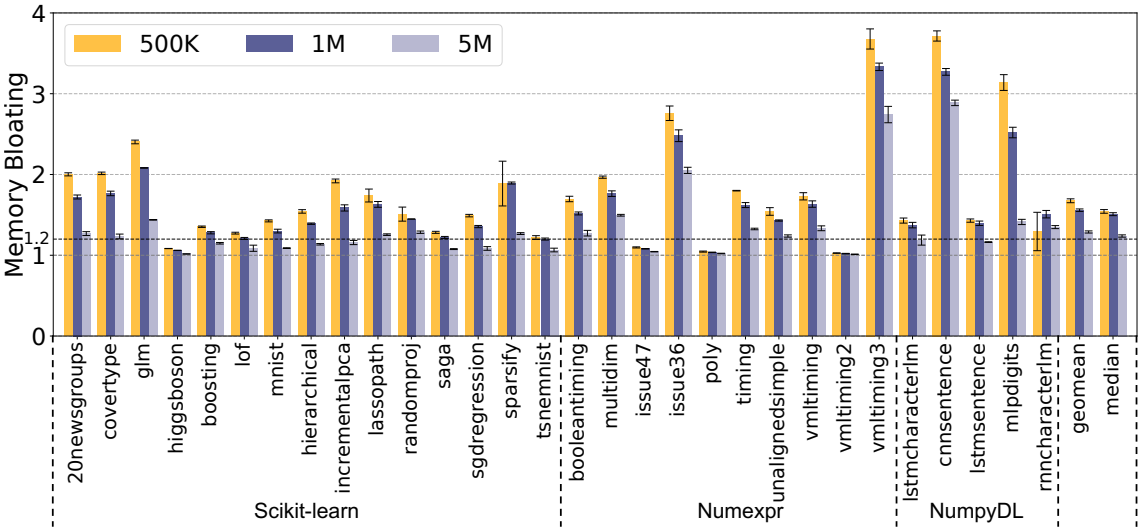
Figure 3.5: Runtime slowdown of PieProf on Scikit-learn, Numexpr, and NumpyDL with sampling rates of 500K, 1M, and 5M. The y-axis denotes the slowdown ratio and the x-axis denotes the program name.

learn benchmark due to space limitations, and exclude `varying-expr.py` from Numexpr, `cnn-minist.py` and `mlp-minist.py` from NumpyDL due to large variations in memory consumption, or the runtime errors of vanilla runs `cnn-minist.py` and `mlp-minist.py`.

We run each experiment three times, and report the average overhead. Furthermore, the overhead of PieProf is evaluated with three commonly-used sampling rates, 500K,



(a) Redundant Stores Detection



(b) Redundant Loads Detection

Figure 3.6: Memory bloating of PieProf on Scikit-learn, Numexpr, and NumpyDL with sampling rates of 500K, 1M, and 5M. The y-axis denotes the slowdown ratio and the x-axis denotes the program name.

1M, and 5M.

Figure 3.5a shows the runtime slowdown of the redundant stores detection. The geo-means are 1.09×, 1.07×, and 1.03× under the sampling rates of 500K, 1M, and 5M, and the medians are 1.08×, 1.05×, and 1.03×, respectively. Figure 3.5b shows the runtime slowdown of the redundant loads detection. The geo-means are 1.22×, 1.14×, and 1.05×,

under the sampling rates of 500K, 1M, and 5M, and the medians are 1.22×, 1.11×, and 1.04×, respectively. The runtime slowdown drops as sampling rate decreases, because more PMUs samples incur more frequent profiling events, such as inspecting Python runtime, querying the CCT, and arming/disarming watchpoints to/from the debug registers. Redundant loads detection incurs more runtime slowdown compared to redundant stores detection, because programs usually have more loads than stores. Another reason is that PiePROF sets `RW_TRAP` for the debug register to monitor memory loads (x86 does not provide trap on read-only facility) which traps on both memory stores and loads. Even though PiePROF ignores the traps triggered by memory stores, monitoring memory loads still incurs extra overhead.

Figure 3.6a shows memory bloating of the redundant stores detection. The geo-means are 1.25×, 1.24×, and 1.23× under the sampling rates of 500K, 1M, and 5M, and the medians are 1.18×, 1.18×, and 1.16×, respectively. Figure 3.6b reports memory bloating of the redundant loads detection. The geo-means are 1.67×, 1.56×, and 1.29× under the same sampling rates, and the medians are 1.52×, 1.51×, and 1.24×, respectively. Memory bloating shows a similar trend to runtime slowdown with varied sampling rates and between two kinds of inefficiency detection. The extra memory consumption is caused by the larger CCT required for the larger number of unique call paths. `issue36`, `vmltiming2`, and `cnnsentence` suffer the most severe memory bloating due to the small memory required by their vanilla runs. PiePROF consumes a fixed amount of memory because some static structures are irrelevant to the testing program. Thus, a program has a higher memory bloating ratio if it requires less memory for a vanilla run. `mlpdigits` consumes more memory for redundant loads detection, because `mlpdigits` (a deep learning program) contains a two-level multilayer perceptron (MLP) that has more memory loads than stores.

Although lower sampling rates reduce overhead, the probability of missing some subtle inefficiencies increases. To achieve a better trade-off between overhead and detecting ability, we empirically select 1M as our sampling rate.

3.6 Threats to Validity

The threats mainly exist in applying PiePROF for code optimization. The same optimization for one Python application may show different speedups on different computer architectures. Some optimizations are input-sensitive, and a different profile may demand a different optimization. We use either typical inputs or production inputs of Python applications to ensure that our optimization improves the real execution. As PiePROF pinpoints inefficiencies and provides optimization guidance, programmers will need to devise a safe optimization for any execution.

3.7 Summary

This chapter demonstrates the *first* to study the interaction inefficiencies in complex Python applications. Initial investigation finds that the interaction inefficiencies occur due to the use of native libraries in Python code, which disjoins the high-level code semantics with low-level execution behaviors. By studying a large amount of applications, we are able to assign the interaction inefficiencies to five categories based on their root causes. We extract two common patterns, redundant loads and redundant stores in the execution behaviors across the categories, and design PiePROF to pinpoint interaction efficiencies by leveraging PMUs and debug registers. PiePROF cooperates with Python runtime to associate the inefficiencies with Python contexts. With the guidance of PiePROF, we optimize 17 Python applications, fix 19 interaction inefficiencies, and gain numerous nontrivial speedups.

Chapter 4

Visual Studio Code in Introductory Computer Science Course: An Experience Report

4.1 Introduction

Integrated Development Environments (IDEs) play an important role in learning a programming language. IDEs offer programmers extensive development abilities. They understand language syntax and provide features such as build automation, code linting, testing, and debugging, which accelerate and simplify the coding process. Through the help of IDEs, students benefit from efficient programming, testing, and debugging. Students can further develop better coding habits and flatten the learning curve of a new language. As a result, more and more instructors start involving IDEs in introductory-level Python courses such as Atom [8], Jupyter Notebook [131, 3, 180], and many others, which significantly improve student's coding experiences.

However, not all IDEs are suitable for introductory-level Python courses. Choosing a satisfying IDE is difficult for instructors. On the one hand, professional IDEs [185, 57, 82] support an integrated programming environment and many powerful features but with

limited support for education. One may need advanced knowledge and plenty of time to use them properly. For students who are new to programming, it is hard to take advantage of IDEs on top of learning a new programming language, because plenty of time is spent on software installation and environment setup. On the other hand, education-focused code editors are easily installed and manipulated, but they are rarely used during professional software development cycles. Students face a big gap when transiting from college to industry [177, 178], which has some negative impacts on their future careers.

Thus, there is urgent to use an appropriate IDE for the introductory-level Python class, which not only provides enough education-related features but also is mainstream among professional software engineers. We aim to use an IDE that satisfies four features: 1) it is cross-platform supported, students are able to install and use it identically on different OS/hardware; 2) it is easy to use, students can code effectively with a simple and concise interface; 3) it has extensive functionalities, e.g. code compilation, project organization, and multi-languages support, etc.; 4) it is mainstream among professional software engineers, which means the IDE is widely used in industry, bridging the gap between introductory-level class and future career.

Thus, we identify Microsoft Visual Studio Code (VS Code) as the desired IDE, which has all four aforementioned features. Moreover, VS Code has been adopted in many advanced courses in our department, such as operating systems, compiler constructions, computer networks, and many others. However, VS Code does not draw significant attention to CS1 courses. Furthermore, with our study of 20 computer science departments, none of them specify VS Code as the default IDE in the introductory Python courses. Moreover, there is no comprehensive guidance on VS Code for educators and students. Since students have various backgrounds, e.g. from diverse majors, with uneven learning speeds, or are familiar with different operating systems, comprehensive guidance is necessary to facilitate the efforts of both students and educators.

In this paper, we give the first experience report for the use of VS Code in an introductory (CS1) Python course. We create the first comprehensive guidance of VS Code for

both students and educators by gathering plugins (i.e., VS Code extensions) for Python education and various user experiences. The guidance is highly modularized. Students who have programming experience jump to the sections they need to learn and skip the sections they already know; students who are beginner programmers follow the guidance step-by-step; students who come across errors target why errors occur and how to solve them. With the help of our guidance, students can quickly acquire the necessary knowledge of the programming environment in the class. We already integrate VS Code and our guidance into the introductory-level Python course in our department. We use a survey to evaluate the effectiveness of our efforts. From the survey, the students highly value the use of VS code and our guidance. Students with some industry internship experiences acknowledge the usefulness of learning Python with VS Code over other education-oriented IDEs, as VS Code is widely used in the industry.

Contributions. In this work, we make the following contributions.

- We point out the importance of selecting a proper IDE for introductory-level courses.
- We identify Visual Studio Code as a desired IDE. We investigate it from four aspects and analyze its practicality for introductory-level Python courses.
- We propose the first comprehensive guidance[1] with hierarchical indexing, to guide students with diverse backgrounds.
- We report our experiences in using VS Code in an introductory-level programming course and show that VS Code is a satisfactory IDE for the introductory-level Python course. We also verify the value and necessity of the guidance.

[1]The VS Code guidance will become public upon this paper's acceptance.

4.2 Background and Related Work

4.2.1 Python in Education

Python has become the most prominent language in college, it tops the list of the most-taught programming languages, especially in introductory computer science courses, it is mainly due to three facts: *Python is beginner friendly*, it has a simplified syntax with an emphasis on natural language while most programming languages have complex rules, thus it is perfect for beginners to learn and practice [90, 128, 156, 40]. *Python is a powerful tool*, it is applied in many areas like machine learning and big data, especially effective and essential in scientific computing and data analysis [121, 179, 127, 28, 27, 165, 1, 125]. Students with diverse backgrounds use Python for different purposes, e.g. analyzing the market in finance, simulating protein structure in biology. Students who pursue a formal education in computer science or computer science-related majors are extremely likely to continue using Python throughout their careers. *Python is popular*, according to the PYPL index [135] and Stackoverflow Developer Survey 2021 [158], Python is the most popular and the fastest-growing programming language (10.4%) in the world as of Jun 2022.

4.2.2 Pedagogical Approaches

Programming language plays an important role in computing education. Many systematic studies explore the teaching in introductory-level programming classes [126, 148, 103, 146].

Through our observation, a tremendous number of research have been done on how the programming environment can improve students' coding experience in recent years, especially the programming environment for Java and Python, which are the most taught entry-level programming languages in college. The selection of the programming environment is typically decided by instructors. Pedagogical tools and Intelligent Tutoring Systems (ITS) [173, 181, 182, 39, 112] have been designed specifically for programming novices. Also, some works report the study of using industry-level programming envi-

ronments such as Jupyter Notebooks [3, 45, 10, 180] or Eclipse extension [142], their experience shows how a great IDE improve students' engagement and performance.

4.2.3 IDEs in Education

Integrated Development Environments (IDEs) refer to software applications that combine all tools needed for a software development project, including an editor, compiler, and debugger. They consolidate different aspects of software development and significantly improve programmers' productivity.

Many types of programming environments exist in the market. Roughly there are two kinds: plain-text/code editor and IDE. Text/code editor does not require complex installation and configuration, some editors are installed by default like NotePad or TextEdit, but they offer limited functionalities and are not directly related to programming. On the other hand, a full-featured IDE combines functionalities in one, integrating all tools developers need to build and test, but IDE requires more disk space/memory or a faster processor, users may suffer more from installation, configuration, even cost (some IDEs' license are expensive). However, there is not a clear boundary exist between them. Users may turn a text/code editor into an IDE by installing plug-ins/extensions. There are always trade-offs between the time spent on installation/configuration and how powerful the functionalities are.

It is important to introduce IDEs in programming courses, especially in introductory-level courses. A great IDE helps students from two perspectives. First, it helps students write correct code. In our past teaching experience, common errors come up over and over again in students who have no or less programming experience. Second, it helps students establish good coding habits. Beginners focus more on correctness and tend to produce low-quality code, including messy code format, meaningless variable names, excessive function length, etc. Many IDEs support features like code formatting, variable name suggesting and function length warning, helping students write clean and decent code that shows professionalism towards industry standards.

4.3 Visual Studio Code Investigation

Visual Studio (VS) Code is a lightweight IDE developed and supported by Microsoft, it is free for private or commercial use. The core feature of VS Code is its extension support, users are able to add languages, debuggers, and tools to their own installation to better serve later development. Besides standard extensions released by Microsoft, there are plenty of extensions on the VS Code Extension Marketplace [107] contributed by third-party organizations and individual developers. We analyze its practicality for introductory-level Python courses from four aspects: accessibility, easy-to-use, functionality, and popularity.

Accessibility. VS Code provides cross-platform support, it runs on various operating systems, i.e. macOS, Windows, Linux, and on most available hardware with an identical user interface. VS Code has a small download (¡ 200 MB) and a disk footprint (¡ 500 MB), it is lightweight, and perfectly fits every student with different devices.

As a multi-languages supported IDE, almost every major programming language has extension support, it is effortless to switch between different programming languages. Though our class is for Python beginners, students can continually code with VS Code in later programming classes.

Easy-to-use. VS Code has a compact but simple user interface. Figure 4.1 shows an example. In the center is the editor, where students code their assignments. Under the editor is the panel, it can be displayed in different panels like a debug console or a built-in terminal, the terminal always starts from the root of a certain workspace. On the left side is the sidebar that contains different views, Figure 4.1 shows the view of Explorer to better assist in locating a file. Maintaining a single window of code and debugging effectively increase programming efficiency.

Besides the concise user interface, VS Code employs many features to improve development. One important feature that simplifies programming is the workspace configure

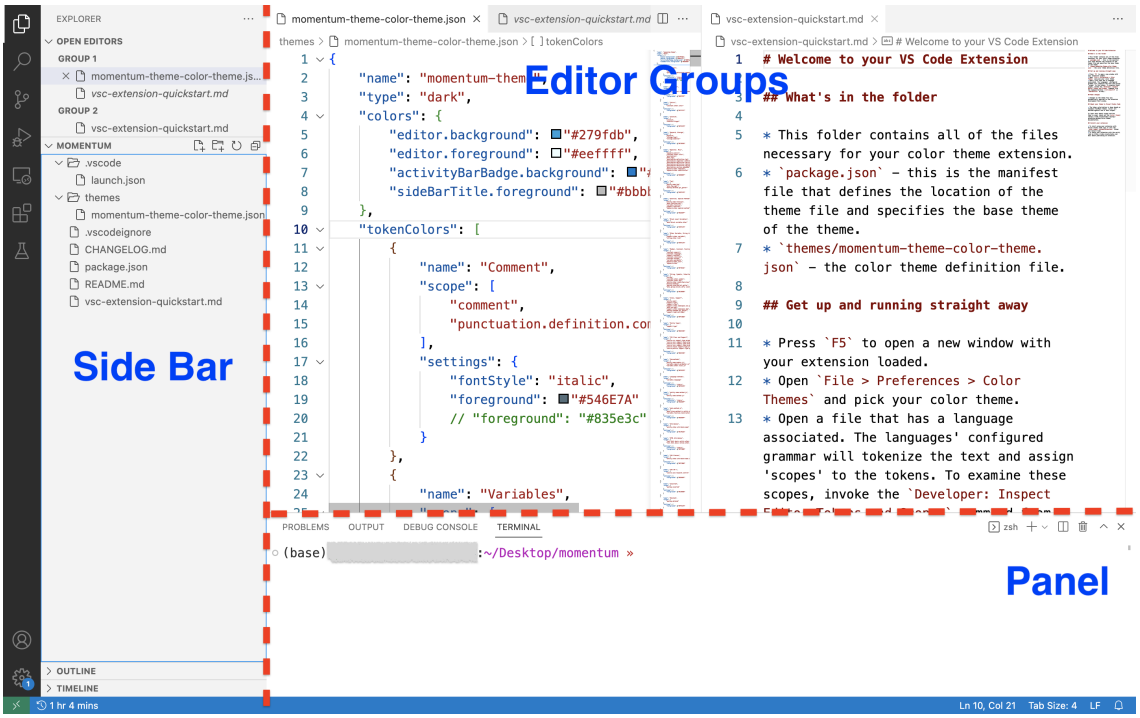


Figure 4.1: The basic layout of VS Code user interface [32] that we introduced to students. The editor is the area to edit files, students can open multiple editors at the same time side by side vertically or horizontally. The panel below the editor is for output or debug information, errors, and warnings, or an integrated terminal. The side bar contains different views like the Explorer or Extension Marketplace, to assist students while working on the projects or downloading extensions.

setting. A VS Code workspace is the root folder of the current project, configure settings that apply to a specific workspace but not others. As a common scenario, students need to have different settings (e.g. interpreter version, dependent libraries, programming languages) among projects. In our past experience, students may feel confused with the configure settings, especially set/reset environment at the beginning of starting a new project. It will be much easier to work on VS Code workspace, because it allows configuring settings in the context of the current workspace, and always overrides the global user settings.

Furthermore, VS Code supports remote development. The *Remote SSH* extension allows opening a remote folder on any remote machine, virtual machine, or container a running SSH server. Another useful feature favored by many developers is the *Com-*

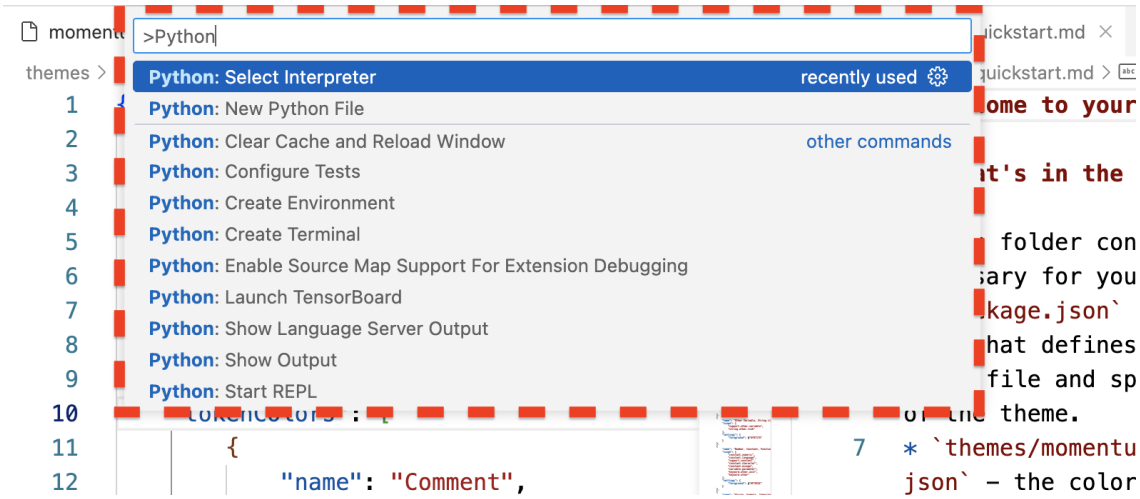


Figure 4.2: The red box indicated the Command Palette of VS Code. Students can access all functionality of VS Code and install extensions through the Command Palette.

mand Palette, Figure 4.2 shows an example, where users would have access to all of the functionalities of VS Code, including commands of your installed extensions.

Functionality. Besides the basic features of a source code editor, VS Code offers extensions to increase its functionality. With over 30,000 extensions in Marketplace, users pick favored extensions and customize their installation to improve the programming experience. For example, in our Python course, we recommend students install the Python extension developed by Microsoft, which offers strong support for Python language, it has powerful features such as IntelliSense [104], code formatting, debugging, variable explorer, and more. The IntelliSense suggestion pops out while you type, it provides intelligent code completion based on the language semantics and written source code. Figure 4.3 shows an example, where IntelliSense suggests using variable `msg` in `print` function.

Popularity. VS Code is widely used among real-world developers. In the Stack Overflow 2021 Developer survey [158], VS Code tops the most popular developer environment tool, with 71.06% of over 80,000 respondents reporting that they use it. Due to the great user amount, bugs/issues are fixed/solved in time. VS Code updates frequently, from June

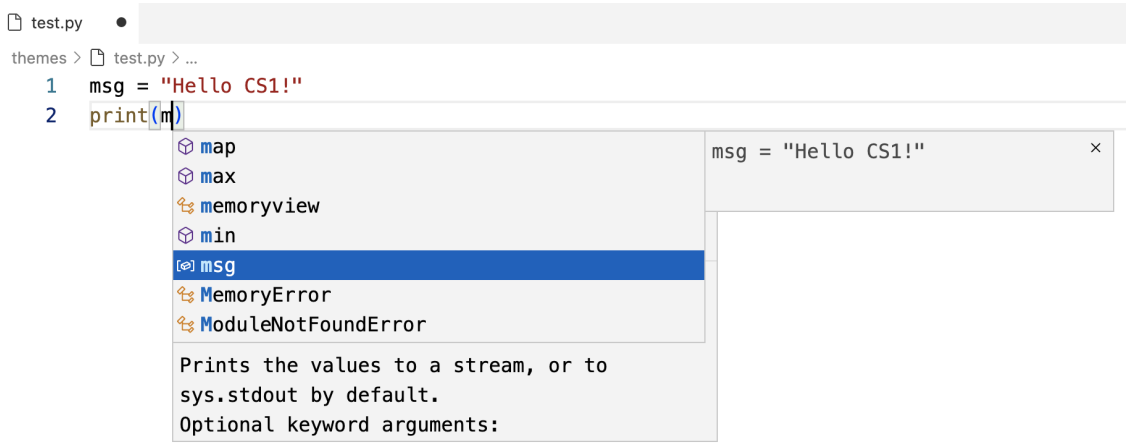


Figure 4.3: An example of code completion supported by VS Code, where the IntelliSense suggests using variable `msg` in `print` function.

2020 to May 2021, VS Code development team made 22 releases in total, updating almost every month.

The high popularity tops the reasons why we introduce VS Code to our Python class. Students encounter difficulties in switching IDEs, they are able to use VS Code throughout different programming languages classes, or in their future careers.

4.4 Guidance and Support

To accelerate the learning process of VS Code for students with various backgrounds, we introduce comprehensive guidance in Python class. The guidance[2] is divided into separate tutorials, the following subsections explain the detail of each tutorial.

Download and installation of VS Code. In this tutorial, students are guided to the download page, and how to choose the correct version according to their OS. We introduce the workspace right after the installation because it is the fundamental unit to manipulate projects. We give examples of how to start a VS Code project, and how to organize files in a project.

[2]Guidance figures are not shown in the paper for anonymous purposes, and the guidance will become public upon this paper's acceptance.

Download and installation of Python. In our past teaching experience, we suffer three main problems with Python version: 1. not being aware of the difference between Python2 and Python3; 2. installing the wrong version; 3. having both Python2 and Python3 installed on the same device, but don't know how to switch them.

In the tutorial, we require students to use Python3 and emphasize the difference between Python2 and Python3. Before the installation, students need to check if they have any versions of Python pre-installed. If they don't have or have Python2 installed, they are led to the newest Python3 download link. If they have any version of Python3 installed, then there is no need to install it again. For students with multiple versions of Python installed, we provide a detailed guide on switching Python versions in the next tutorial.

Setup Python environment. Next, students install the Python extension and set up the Python environment in VS Code. In our past teaching experience, students were always confused with the programming environment settings, we pay extra effort into where and how to set/change configure settings.

Then students are guided to the extension marketplace, to find and install the Python extension developed by Microsoft. The Python extension works on/with any operating system/Python version, the Jupyter extension is included in the Python extension installation bundle. The Python extension supports code completion and IntelliSense on top of the currently selected Python interpreter, we provide step-by-step guidance with figures on how to select a Python interpreter from *Command Palette*. We also provide guidance on how to change programming language mode, for students who want to work with other programming languages in later classes.

Run Python examples. Once set up Python environment, VS Code becomes a real Python IDE. In this tutorial, we guide students to write, run their first simple Python program from scratch, and give explanations on how Python extension improves coding.

We require students to generate expected outputs for example programs, such that they have identical settings for the Python environment.

Install and run Jupyter extension. We use VS Code's Jupyter extension to run in-class coding exercises to improve student engagement [131, 3], through the Jupyter extension students open and run ipynb files on VS Code the same way as Python files. The Jupyter Notebook extension provides basic notebook support for language kernels and allows any Python environment to be used as a Jupyter kernel. In this tutorial, we guide students to install the Jupyter extension, create new Jupyter files, and manage and run the code cells.

Install and use scientific packages. Python packages are very efficient to solve complex problems in scientific computing, data visualization, data manipulation, and many other fields. In this tutorial, we guide students to install and import Python libraries, e.g., `Numpy`, `Pandas`, `Matplotlib`, and `SciPy`. Then students are required to learn and use basic APIs, and the example codes are provided. To succeed in this tutorial, students need to plot a certain figure using `Matplotlib` functions.

4.4.1 Improvements for Guidance

To better guide students, the guidance comprises optimizations as follows:

- We provide two versions of guidance for MacOS users and Windows users, respectively. In our experience, a single version of guidance for all OSs is not adequate, minor differences among OSs matter in the introductory-level course.
- We provide three ways to access guidance: PDF download, website, and video. The conventional way is to upload guidance to the course learning management system as PDF, but its format (PDF) has limited representations, therefore we add a website version of the guidance. Due to COVID-19, students might not be able to get

enough in-person instruction, we also provide video guidance to improve student engagement.

- We constantly collect students' feedback, and periodically improve and update the guidance.

4.4.2 Hierarchical Indexing

We provide detailed descriptions of tutorials, students follow tutorials step-by-step, and set up a programming environment without effort. However, students' preferences regarding tutorials vary with their programming backgrounds. Tutorials with a flat structure cannot satisfy all students' requirements. Students who prefer video tutorials may not follow the video pace from the beginning. A common scenario is to start with a verbal demonstration and refer to a video when problems occur. It is difficult to locate the corresponding video timestamp from text if tutorials are flat structured. In addition, students with some programming knowledge require more concise messages, e.g. have installed Python and VS Code, but require guidance on Python environment in VS Code, which fail to provide if tutorials are flat structured. For students who are already familiar with the environment setup, it is hard for them to extract key messages from flat-structured tutorials.

We design hierarchical indexing on tutorials to fulfill diverse requirements. The hierarchical indexing consists of three levels: 1) High-level: we provide an abstract with all key messages at the beginning of each tutorial, such as software names, download links, software versions, etc. 2) Medium-level: we segment each tutorial into several parts and summarize with subtitles. We list subtitles and link the corresponding positions in tutorials. Students can locate the information they need without effort. 3) Low-level: we build mappings between video and verbal demonstration, i.e., verbal demonstrations for each step within a subtitle are linking the corresponding video timestamps. Thus, students switch between verbal and video demonstrations when problems occur.

4.5 Evaluation and Student Responses

In this section, we evaluate two objectives:

- Validate VS Code is a suitable IDE for the introductory-level Python programming courses
- Verify the value and necessity of VS Code's Guidance

4.5.1 Course Description

We introduce VS Code and VS Code guidance to a CS1 Python programming course in Fall 2022 and Spring 2023 semesters, the course enrollment is 141 and 93, respectively. This CS1 course is restricted for non-CS students, with an emphasis on basic programming skills and engineering applications. Students learn how to solve problems through writing Python programs, particular elements include: the development of Python programs from specifications; documentation and coding style; use of data types, control structures and data structures; abstractions and verification.

The course uses a flipped classroom format. Every week, students start conceptual learning of the topics covered in the week by watching videos and completing self-check quizzes online. Then they attend a 100-minute class exercises session and a 165-minute lab exercises session to participate in the active learning activities. Students work on weekly homework assignments and project tasks after class to reinforce the knowledge and skills of the week. Students are required to run the in-class coding exercises via the Jupyter Notebook extension installed in VS Code, and complete the programming tasks (homework assignments, lab exercises, and projects) in VS Code by themselves.

4.5.2 VS Code and VS Code Guidance Evaluation

We designed surveys with 21 questions, to collect students' feedback on the VS Code and VS Code Guidance at the end of the semester. The survey first collects students'

backgrounds, then invites students to rate both VS Code and VS Code guidance based on their programming experience throughout the semester. In total, 82 valid responses were collected, 42 responses in Fall 2022, and 40 responses in Spring 2023.

First, we collect students' background information. In Fall'22/Spring'23, 79%/83% of students are freshmen and sophomores, 86%/88% of students did not take any Python-related programming course in the past, and 74%/75% of them have less than one year of coding experience.

Then students are invited to rate the VS Code and VS Code guidance performance throughout the semester. They rate VS Code from the following four aspects, the degree of satisfaction lies between 1 to 5, in which 1 is strongly dissatisfied and 5 is strongly satisfied.

- Visual appeal: rate the VS Code's user interface, and the editor layout.
- Extension ecosystem: rate the way to search/install/uninstall extensions.
- Debugging experience: rate the VS Code's built-in debugger, whether it helps accelerate students' edit, compile, and debug loop, and if the recommended debugger extensions are helpful.
- Editing experience: rate the overall editing experience of VS Code. If the basic editing features (i.e., keyboard shortcuts and Command Palette) are useful and beginner-friendly. Also rate IntelliSense, if code editing features such as code completion, parameter info, and content assist are helpful.

Based on the responses, in Fall 2022 semester, 61% of students do not have IDE or coding platform experience in the past, and 39% of students have used some other IDEs, whereas in Spring 2023, 70% of students do not have IDE or coding platform experience in the past, and 30% of students used some other IDEs such as Matlab, RStudio, etc.

As shown in Table 4.1, the average rating for visual appeal, extension ecosystem, debugging experience, and editing experience are 4.17, 3.81, 4.02, and 4.05, respectively

Table 4.1: The averaged answers of students' satisfaction (82 students in total) over four aspects: visual appeal, extension ecosystem, debugging experience, and editing experience. The degree of satisfaction lies between 1 to 5, in which 1 is strongly dissatisfied and 5 is strongly satisfied.

Semester	Visual Appeal	Extension	Debugging	Editing
Fall'22	4.17	3.81	4.02	4.05
Spring'23	4.28	3.71	3.7	4.73

in Fall 2022; and 4.28, 3.71, 3.7, 4.73, respectively in Spring 2023. We released a VS Code extension that periodically recommends and installs themes for students, which significantly raises their editing experience[3]. As for the student's overall satisfaction with VS Code guidance, it is 4.2 in Fall 2022 and 4.0 in Spring 2023.

Among the total 82 students, 74% students consider VS Code easy to install and use with the help of VS Code Guidance, and 76% students consider VS Code as a good IDE to code with. There are 13 students who claim they had issues/trouble with VS Code guidance throughout the semester, and we have collected these issues and fixed/updated them in the VS Code guidance.

4.5.3 Class Averages

We also compare the class averages of Spring'22 with Fall'22/Spring'23, the results are shown in Table 4.2. In Spring 2022, the class average total score is 82.86%, whereas 85.10% and 84.34% in Fall 2022 and Spring 2023, respectively.

Table 4.2: The class average of total score in the Spring 2022, Fall 2022, and Spring 2023 semesters.

Semester	Spring'22	Fall'22	Spring'23
Class Average	82.86%	85.10%	84.34%

[3]We collected students' feedback but the data is not shown here since it has been discussed in another work.

Although the comparison in Table 4.2 shows that the class averages of Fall 2022 and Spring 2023 are higher than Spring 2022, we cannot simply accredit the average raise to VS Code or VS Code guidance. In Spring 2022, we used the traditional classroom, and students are required to use Spyder integrated within Anaconda, whereas we use the flipped classroom and VS Code in Fall 2022 and Spring 2023. But according to the student's experience, we can conclude that both VS Code and VS Code guidance have positive effects and are useful in completing coding assignments.

4.5.4 Student Comments

At the end of the student survey, students are invited to comment on their VS Code and VS Code guidance experience throughout the semester. One student described his/her overall experience as:

> I can view Python files and text files alike without even making a new window using VS Code. The interface is visually appealing. The VS Code tutorial is straightforward. Instructions are easy to follow. That's all a tutorial really needs, in my opinion.

Some students favor the dark mode in VS Code, one student claimed it improves his/her coding experience:

> I like VS Code uses color coordination to make different code types and functions stand out. I also like that by default the background is black it makes it easier to read. 90% of the time it's very easy to find mistakes within codes. Overall the tutorial is easy to follow.

We observe that many students prefer dark mode compared with bright mode, it may be because dark mode or dark theme stands out the syntax highlighting. With a light background, the code texts are mostly darker colors, while more colorful with a dark background. In addition, compared with pedagogical IDEs, changing themes in VS Code is quite easy. Students highly raise the theme extensions in VS Code, one student claimed:

> I like the way how VS Code installs extensions, I especially love theme extensions in Marketplace, whenever I am bored with codes I change my theme, and it just becomes a totally new software!

From the feedback, we collect many constructive suggestions. Some students complain about the complex commands on the terminal. It's true that we may involve plenty of terminal instructions at the beginning of class, but the instructor thinks those instructions/commands are required for this class. To fix this problem, we divide the command study into different labs and add detailed explanations on every instruction.

Another student considered VS Code guidance useful, but suggested we could involve more helper extensions in class:

> Tutorials are thorough, straight to the point. But in my opinion, VS Code lacks collaborative elements, especially in the lab. I feel that collaboration via a 'shared' document would be beneficial because, without it, one user mostly does the code by themselves.

The live share extension enables students to share screens and collaborate with classmates/TAs/instructors on the same code without the need to sync code or configure the same development tools, settings, or environment [106]. We are evaluating the feature and will introduce it to students after careful consideration.

To summarize, students' overall experience with VS Code and VS Code guidance is positive and encouraging, which shows that VS Code and guidance are valuable and promising for the CS1 course.

4.5.5 Issues with Jupyter Notebook in VS Code

We observe two issues: 1. when running a cell/all cells in Jupter Notebook file, it loads for a long time but not showing any content, this issue was reported by more than 10 students (with both on Mac/Windows devices); 2. when running a cell/all cells in Jupyter Notebook

file, VS Code crashes. With 2 students these issues are solved by restarting/upgrading VS Code, but in most cases, we can not fix these, running cells bloat device memory, we fix these by using the Jupyter Notebook web-based platform instead of the Jupyter Notebook extension.

4.6 Discussions

Discussions on education-related concerns. VS Code has some advanced features that raise some debates for their use in introductory-level programming courses, such as code auto-completion and function signature hints. Some argue that supplementary features would develop a blind dependence on IDEs rather than truly understanding the code. In practice, our teaching plan covers all necessary programming knowledge and concepts, and features like auto-completion and function signature hints help students learn faster and more efficiently. Moreover, these features can be customized or completely disabled in VS Code if required.

Discussions on some limitations. We can see some limitations in this study. We only evaluate an introductory Python course. The reason is that our department offers Python as the introductory programming language. However, VS Code supports multiple programming languages, such as Matlab, Java, and C/C++. It is straightforward to adapt our guidance to courses with other programming languages. We will partner with instructors in our department to report more VS Code experiences.

4.7 Summary

This chapter describes the experiences of introducing Visual Studio Code in an introductory (CS1) Python programming course at a big engineering university. In this paper, we investigate VS Code as a satisfactory IDE for CS1 programming courses. To better help students, we develop comprehensive VS Code guidance for students with various program-

ming backgrounds. We perform evaluations among students and validate the practicality of VS Code and verify the quality of our VS Code guidance. We periodically update and improve the guidance with the collected feedback. We are now practicing VS Code and our VS Code guidance in more CS1 programming courses with programming languages besides Python.

Chapter 5

ProTracker: Estimating Progress of Programming Assignments

5.1 Introduction

It is common knowledge that many students struggle with programming assignments, especially CS1 students with limited or even no coding experience. While instructors and teaching assistants (TAs) provide various resources to flatten learning curves, students still face many difficulties, such as inadequate time to complete coding assignments, hard to debug by themselves, shortage of instructor/TA office hours, and many others. To succeed in programming assignments, students need to produce high-quality codes and deliver assignments on time. There are many well-studied approaches to help students write high-quality code such as involving debugging or testing frameworks [11, 9, 160, 12, 155], but not many researches have been conducted on how to help students estimate the assignment workloads and deliver assignments on time. Without such a tool, students may fail to control the pace of finishing the assignments and miss the deadlines.

To avoid such cases, instructors encourage the students to start the assignment early and provide more leniency, like pushing back deadlines and offering extra office hours. However, these approaches are usually unproductive, as the students have other deadlines

and may rank this assignment with lower priority. Thus, a fundamental solution is to let students self-manage their time better, so they do not struggle when the deadline is approaching. To achieve this goal, it is important for students to understand their pace toward completing assignments by tracking their own progress. We foresee a tool that reports positive and accumulated progress can benefit both students and instructors/TAs.

From the student's perspective, tracking their programming progress gives them better time management to complete coding assignments more efficiently. Moreover, positive feedback (i.e., a growing progress bar) motivates students towards completing the assignments. We survey a number of students who enrolled in a CS1 Python programming course in Summer 2022, of whom 85.3% students (29 out of 34 students) would like to know their programming progress to better estimate the assignment completion time. From the instructor's aspect, understanding students' programming progress helps them identify any learning obstacles to studying (i.e., many students are stuck at a certain progress status for a long time), so they can provide necessary help accordingly.

However, having students estimate their assignment progress is challenging. Programming assignments typically require students to implement one or multiple functions, data structures, or algorithms, which consist of many components. Students, especially those who take the CS1 courses, can mistakenly estimate the required efforts: underestimation results in the risk of not finishing the assignment by the deadline, and overestimation leads to the impression that the assignment is over complex. Thus, there is a demand for an automatic method to help students estimate programming progress accurately.

The software engineering community has some prior work on better estimating or tracking software project progress [4, 34, 5, 175]. However, prior approaches are developed for engineering managers or architects, not for educators or students. Estimating the progress of programming assignments in the education community has its challenges and opportunities. First, estimating software project progress in prior works requires involving multiple aspects including budgets, software design, and communication, among others. In contrast, we only focus on students' assignments. Moreover, we are able to

refer sample solutions for progress estimation but prior works can only refer to the system design "blueprint", which is not detailed and accurate. Furthermore, we have plenty of assignment samples (provided by previous students) as references to extract common patterns for progress estimation, while prior works target individual software projects.

Given these opportunities, the research challenge here is how to use sample solutions and assignment samples to estimate programming progress for students tracking their assignment progress, without giving any hints for problem-solving. This paper targets the research question: *How can we estimate the programming progress of assignments for educational purposes?*

To answer this research question, we propose ProTracker, the first end-to-end solution to estimate the coding progress of programming assignments for students by applying machine learning techniques. ProTracker produces accurate programming progress estimation on-the-fly while doing assignments, so students can get real-time feedback. Figure 5.1 shows a typical pipeline for students doing a programming assignment. When starting an assignment, students first do coding for several milestones or the entire program. They then test their codes and do debugging if necessary. These patterns will repeat until the final solution is submitted. As an extension, ProTracker can be easily installed on students' editors[1], upon installation, the estimated programming progress appears whenever students save code texts. ProTracker tracks the coding progress for the entire assignment or milestones only regardless of the code correctness. For correctness, ProTracker incorporate with testing/debugging approaches [9, 160, 11, 75]. Students can seek help from the instructor/TAs in time if they (1) feel their coding paces fall behind or (2) complete the coding progress but are stuck at debugging for a long time.

ProTracker consists of two parts: backend and frontend. The backend takes sample solutions and assignment sample files as input, preprocesses the code text, performs static analysis, and generates the dataset for ML models. It applies a two-level cross-validation method to perform hyper-parameter tuning for each model and select the best model for

[1]We provide step-by-step installation instruction, which normally takes less than a minute.

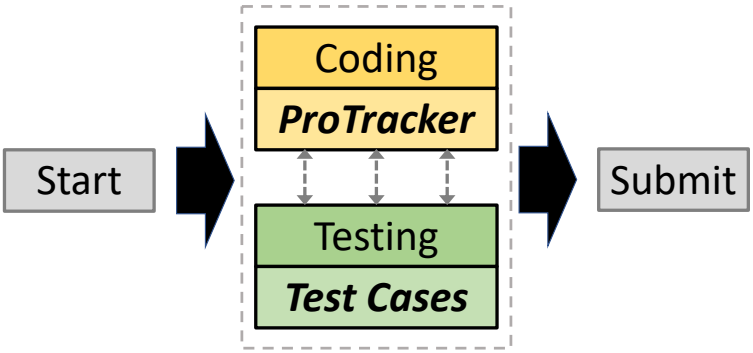


Figure 5.1: How we help students in code assignments. We provide test cases to check code correctness, and ProTracker to estimate coding progress.

the input data. After the backend produces the best ML model, the frontend, which runs as a Microsoft Visual Studio Code (VS Code) extension (i.e., the default programming environment in our courses), uses the model to estimate the progress and report to the students on-the-fly.

Contributions. In this work, we m

ake the following contributions.

- We bring up the necessity of measuring and reporting the progress of programming assignments to both students and instructors.
- We propose a machine learning-based approach to estimate the progress of programming assignments with high accuracy and low overhead.
- We develop an end-to-end system, ProTracker, to show the progress status (in percentage) in the editor, which can be used in practice.

5.2 Related Works

5.2.1 Predictions in Education

In recent years, tremendous research has focused on student behavior predictions with educational data mining techniques. Works have been done on student success prediction, typically they collect data (e.g., students' background information, assignments scores, and grade point average (GPA) within the intended major) from the early stage of class to predict students' final grades or performances [138, 130, 152, 16, 54], some other works [18, 184, 188, 193] reported significantly high prediction accuracy but not able to predict in the early stage of the class. Furthermore, some researchers have been working on the predictions of student engagement [52, 111, 109, 170], they study the student activity, such as keystrokes and elapse time, to predict students' engagement while programming.

5.2.2 Progress Monitoring in Education

Monitoring student progress plays an important role in education, typically refers to techniques that assess students' academic performance during semesters [96]. Instructors spend a significant amount of time monitoring the progress students made, if and how they can achieve success in class. To accurately evaluate students' progress, first instructors ensure the students' background and current performance level, and the achievement goals students suppose to reach by the end of the learning period. Studies point out that closely monitoring student progress has a positive effect on both instructors and students [145, 194], and adaptive feedback motivates and improves students' performance [101]. Automatic unit tests [11, 9] can help students understand their milestones. However, unit tests can only validate the progress when a milestone completes. ProTracker, instead, estimates the progress at any time.

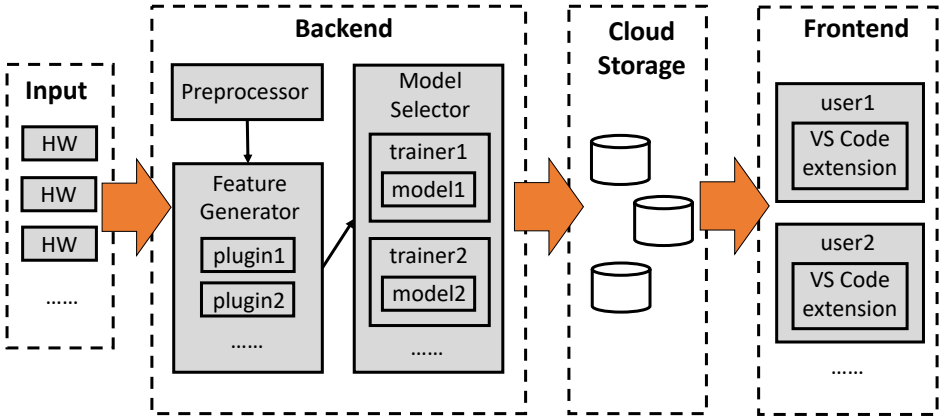


Figure 5.2: The overview of ProTracker. It takes historical students' programming assignments as inputs and achieves programming progress prediction through the backend and frontend, which interact with cloud storage.

5.3 Design and Implementation

Figure 5.2 shows the overview of ProTracker. Instructors, i.e., tool operators, collect historical students' assignments as ProTracker input data, then ProTracker's backend uses input data to train ML models and upload them to the cloud storage automatically. Students, i.e., tool users, install VS Code and ProTracker extension on their local programming environment. ProTracker's frontend (a VS Code extension) downloads the updated ML models from the cloud and predicts students' programming progress on-the-fly. The pre-trained ML models can be shared among different instructors, but we highly recommend instructors retrain the existing models with their own input data (historical students' assignments from their courses), for optimal accuracy.

The backend runs on the server and consists of three major components: *(i) Preprocessor:* the preprocessor takes collected labeled assignments as input data, and generates clean and valid code text. *(ii) Generator:* generator performs static analysis on the clean code text. It extracts labels with predefined patterns and produces feature data based on pre-configured plugins. With the labels and feature data, it generates datasets for the model selector. *(iii) Model Selector:* the model selector applies datasets created by the generator on hyper-parameter tuning for each ML model. It picks the best model for the

input files and uploads the selected model with tuned parameters to the cloud storage.

The frontend runs as VS code extension that is installed on users' editors. The frontend pulls the updated models from cloud storage when the user opens the VS code. During programming, the moment the user saves a file the frontend performs static analysis on the valid code text, extracts the feature data, and estimates the programming progress with the ML model. The results appear as a percentage number on the user's VS code workspace.

5.3.1 Input Dataset

The core of machine learning is to learn with historical data and predict a selected property of the data for new inputs. The inputs for ProTracker are students' historical programming assignments with pre-defined labels. The collected assignments are labeled by the labelers (i.e., instructors and TAs), who fully understand the assignments. On the basis of assignment grading criteria and sample assignment solutions, labelers are able to label progress for assignments intuitively even if they have diverse solutions. Figure 5.3 shows an example of a labeled assignment file, where the pre-defined label pattern is "#@#", labelers divide the assignment into four parts with 25% of each. By learning the input data (labeled historical students' assignments), ProTracker is able to predict programming progress for new inputs (the code text that students are working on).

Figure 5.3: An example labeled assignment.

5.3.2 ProTracker Backend

Preprocessor. The preprocessor takes students' labeled homework files as inputs, then loads contexts from inputs, performs preprocessing, and produces clean and valid code text for the generator. These files are labeled with pre-defined labels, providing context for ML models to learn. Based on labeling criteria such as homework grading criteria, the logic of sample homework, and test cases, labelers add labels to indicate different percentages in a homework file.

There are three steps in preprocessing. 1) Preprocessor cleans the code text for privacy purposes. It's common to list students' private information (e.g. student name, student id, submission date, etc) in code text, which should not be exposed to the tool's operators. The preprocessor removes the privacy information before the operator access the code content. 2) The code contexts that are not related to the programming functionality (such as empty lines, comments, or pre-defined functions) bring the noise to the ML models' training and result in low accuracy. The preprocessor deletes noisy contexts from the code text, then the generator is able to generate high-quality datasets, improving the accuracy of ML models. 3) There may have invalid label errors and syntax errors in the code text, which trigger errors during the generator's static analysis. The preprocessor validates the code text and fixes/removes the invalid code blocks from it, which avoids errors during the generator producing the datasets.

Generator. The generator extracts the labels and produces features based on the code text delivered by the preprocessor, and generates formatted datasets for the model selector. When the generator generates the data for the code text of a file, it first locates all labels $\{l_1, l_2..l_K\}$ in this file by using a predefined pattern with a pattern match. Based on the total number of labels, the generator gets the real value of each label by proportional calculation: $l_i = \frac{i}{K}$. As the example shown in Figure 5.3, the value of the first label $l_0 = 25\%$ and the value of the second label $l_1 = 50\%$, ..., so on and so forth. For each label, the generator segments and identifies the code block, and generates its corresponding

feature data by performing static analysis with feature extraction plugins. After producing features for all labels, the generator formats, concatenates, and compresses all the data and passes them to the model selector for ML models' training. Besides the default features, ProTracker provides APIs for the tool operators to change the feature produced by developing and loading their feature extraction plugins.

Model selector. The model selector trains and picks the best model for the dataset passed by the generator with a novel two-level cross-validation method [140]. The model selector first split the dataset D into two equally: a dataset for hyper-parameter tuning (D_{ht}), and another dataset for model selection (D_{ms}). Then the model selector performs bottom-level cross-validation. It divides $(\mathbf{x}_i, y_i) \in D_{ht}$, $i \in [1, N]$ to K folders as $\{d_{ht}^1, d_{ht}^2, .., d_{ht}^K\}$ and $\kappa : \{1, 2, ..., N\} \to \{1, 2, ..., K\}$ is the index mapping function between samples to folders. For each candidate model $m_i \in \mathbb{M}$ ($i \in [1, T]$) with L parameters, its estimator $\hat{m}_i^{\Theta}$ depends on tuning parameters $\Theta = (\theta_1, \theta_2, ..., \theta_L) \in \mathbf{R}^L$. For the kth folder, the model selector trains the model to the other $K-1$ folders and calculates the prediction error on d_{ht}^k. The corresponding estimator of kth folder denotes as $\hat{m}_{i,-k}^{\Theta}$. For each tuning parameter θ of m_i, the cross-validation error is:

$$\mathbf{Erf}_{bottom}(\theta) = \frac{1}{N} \sum_{j=1}^{N} (y_j - \hat{m}_{i,-\kappa(j)}^{\theta}(\mathbf{x}_j))^2 \tag{5.1}$$

The model selector finds $\hat{\theta}^i$ for each candidate model m_i, which satisfies:

$$\hat{\theta} = \underset{\theta \in \{\theta_1, \theta_2, ..., \theta_P\}}{\operatorname{argmin}} \mathbf{Erf}_{bottom}(\theta) \tag{5.2}$$

After the bottom-level cross-validation, the model selector gets T tuned models: $\{\hat{m}_1, \hat{m}_2, ..., \hat{m}_T\}$. Then the model selector performs top-level cross-validation to select the best model among candidates. Same as the bottom-level cross-validation, the model selector divides $(\mathbf{x}_i', y_i') \in D_{ms}$ to K folders and gets the index mapping function κ'. The cross-validation error for the t^{th} candidate is:

$$\mathbf{Erf}_{top}(t) = \frac{1}{N}\sum_{j=1}^{N}(y'_j - \hat{m}_{t,-\kappa'(j)}(\mathbf{x}'_j))^2 \tag{5.3}$$

And the model selector picks the $\hat{m}_{\hat{t}}$, which has minimal cross-validation error as the best model for the input dataset.

After the execution of two-level cross-validation and having the best model, the model selector uploads the selected model with its tuned hyper-parameters to the cloud storage, thus the frontend is able to use it for forecasting.

5.3.3 ProTracker Frontend

The frontend presents as a VS code extension. Students download, install and manage it without effort. The extension is activated after VS Code starts up. Upon installation, the frontend uses a pre-trained model that is provided by the backend and performs a real-time estimation for programming progress. The frontend updates the local model with the newest hyper-parameters downloading from the cloud storage automatically without updating the extension. The frontend consists of two parts: a Typescript driver and a Python processor. The driver registers callbacks on saving file events to VS code, and callback functions are executed when students save the file (`onDidSaveTextDocument`). Callback functions load the code text from the alive (the current valid) workspace and pass it to the processor. Then the processor achieves the core functionality of the frontend — programming progress estimation by three steps: (1) validate the input code text (2) apply static analysis to the code text and extract the feature data (3) get the prediction from the pre-trained model with feature data. At last, the processor returns the prediction of programming progress to the driver, and the driver shows it as a percentage of the student's workspace. Figure 5.4 describes an estimating example of frontend, the orange box shows the start estimating message, which pops out when the estimating starts. The red box in the status bar indicates the estimated percentage, which reports the real-time programming progress percentage based on the current code text.

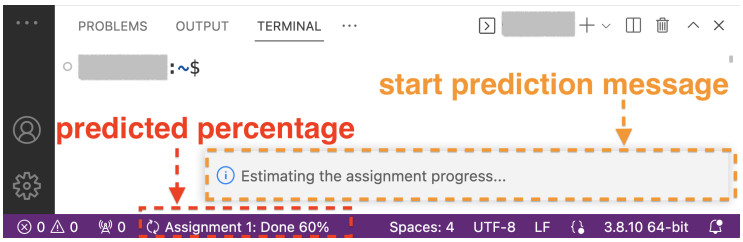


Figure 5.4: An example of real-time programming progress prediction performs by the frontend. The start estimating message and progress estimation appear whenever students save code text.

Different from the backend, it is highly possible that the students' code text has syntax errors during the programming. To solve this problem, two strategies are applied on the frontend: (1) if syntax errors occur at the bottom part of the code text: it's more possible that the student is adding context. The frontend cuts off the code text after the syntax errors and predicts the progress with only the correct code. (2) if syntax errors occur at the top or middle part of the code text: it's more likely that students are modifying the existing context. In this scenario, the frontend simply uses the last result without performing a new prediction. We further optimize the frontend by preventing unnecessary forecast when: (1) multiple files save events occur in a short time window and (2) the code text has no significant difference between multiple successive events (e.g. only one variable name or value has been modified).

Table 5.1: Topics and learning outcomes of corresponding assignments in our introductory-level Python programming course.

Topics	Learning Outcomes	Assignments
Fundamental Programming	Use Python `input()`/`print()` functions to process input/output	Lab1
	Build expressions using arithmetic operators	Homework1, lab2
	Convert data types among `int`, `float`, and `str`	Homework2
Functions	Create user-defined functions	Lab3, Homework3
Conditionals	Write programs using chained/nested conditional statements	Lab4, homework4
`while`/`for` Loops	Implement `while`/`for` loops	Lab5, homework5
Objected-Oriented Programming	Use Python classes and methods	Lab6
Strings and Lists	Create and use 1D/2D lists, traverse lists, convert between list and string	Lab7, homework7, lab8, homework8
Tuples and Dictionaries	Create and use dictionaries	Lab9, homework9
Files and Exception Handling	Read data from text files, handle potential exceptions in the program	Lab10, homework10
Python Scientific Libraries	Create and use NumPy arrays	Lab11, homework11
Plot with Matplotlib	Plot data from text files, plot fitted curves in regression analysis	Lab12, homework12

5.3.4 Discussion

In this section, we discuss the details of ProTracker's implementation.

Sequential labeling. As mentioned in Section 5.3.1, the input data of ProTracker's backend are students' homework assignment files, which are sequentially labeled. With the sequential labeling method, a new label represents a sequential increment of programming progress, which may not be true for software development. For example, four functions exist in a file sequentially as $\{f_1, f_2, f_3, f_4\}$, then the file is labeled as $\{f_1, l_1, f_2, l_2, f_3, l_3, f_4, l_4\}$. Under the situation of the sequential labeling, labels $l1, l2, l3, l4$ represent the progress of 25%, 50%, 75% and 100%, respectively, which indicates the developing order is f_1, f_2, f_3, f_4. However, the real developing order could be f_1, f_2, f_4, f_3 (f_3 calls f_4 in the body), and the sequential labels cannot reflect the correct progress. Nonetheless, we still apply the sequential labeling for ProTracker based on two reasons: (1) it's unrealistic to track every student's programming process in labeling. (2) compare to the developing order, the code text content is more related to the programming progress. Although the sequential labeling may not reflect actual programming progress, it's capable of producing high-quality training data in practice.

Feature selection. We select line number, defined variable number, keyword number, and operator number as features to predict the programming progress. These features are intuitively related to programming progress, but they cannot be used standalone for prediction. For example, more lines of code indicate more programming progress, but it is inaccurate since line numbers vary in coding styles and implementation methods. Thus we combine features to train and forecast. We add/switch features by using feature extraction plugins. Due to the IRB restriction, we have a limited number of samples (students' assignments files), thus we limit the feature number (feature dimension) below five. With the small sample size, high dimensional data make models hard to converge.

Models. ProTracker contributes four built-in candidate models, *Liner Regression* [149], *KNN* [132], *MLP* [70] and *GBRT* [25] to predict programming progress for Python assignments. The reasons for not involving complex machine learning models are: (1) the complex models are harder to converge with a small sample size; (2) lightweight models fit well enough due to the low dimensions and high signal-to-noise ratio of the dataset; (3) complex models have higher forecasting overhead.

Typically, one semester consists of multiple assignments. There are two ways of training: use a single assignment to train individually (each assignment has one model), or use all assignments to train (all assignments share one model). On the one hand, training with a single assignment has higher accuracy but models are easily overfitting [47] and hard to tune with a small sample size. On the other hand, training with all assignments significantly deteriorates the accuracy even applying normalization for features. In practice, we combine multiple assignments that share a similar structure, achieving a good trade-off between accuracy and model tuning difficulty.

Table 5.2: The sample size for six datasets.

Dataset	1	2	3	4	5	6
Sample Size	2226	1655	1359	4571	2009	4304

5.4 Evaluation

This section tests ProTracker on real-world datasets collected from a Python programming course and evaluates ProTracker on programming progress prediction accuracy and runtime overhead.

5.4.1 Datasets

Course description. To build datasets, we collect participants' assignment files from two semesters of a Python programming course. Participants are full-time students who enrolled in an introductory-level Python programming course from a large engineering

Table 5.3: Forecast results for each dataset.

Dataset	1	2	3	4	5	6
Picked Model	GBRT	GBRT	KNN	KNN	GBRT	GBRT
MSE	0.0048	0.0058	0.0117	0.0139	0.0192	0.0084
R2	0.94	0.92	0.84	0.82	0.77	0.89

university, in Fall 2021 and Spring 2022. This course is prepared for students not majoring in Computer Science. The course enrollment amount is 136 students for Fall 2021 and 86 students for Spring 2022.

Assignment description. Each semester, there are 11 homework assignments and 12 lab assignments (3 individual lab assignments, 9 group lab assignments), in total 23 assignments with 4325 files, the detailed assignment description is shown in Table 5.1. We group assignments that have similar programming structures and generate six datasets, the sample size of each dataset is shown in Table 5.2.

5.4.2 Accuracy

Evaluation setup. We perform an end-to-end evaluation of ProTracker's accuracy by performing a breakdown analysis on the two-level cross-validation of the model selector. For each dataset, we first split the dataset into training/testing sets by 80%/20%. Then we perform a 50%/50% split on the training set, generating D_{ht} for hyper-parameter tuning of candidate models and D_{ms} for best model selection. We use five as the folder number (K) for cross-validations [183]. For hyper-parameter tuning, we pre-define the search space and max iteration to find the local optimal hyper-parameter combination. We present the ProTracker accuracy by two metrics: mean squared error (the lower the better, 0 is the best), and R^2 score (the higher the better, varies between 0 and 100%), they are both reported as the average value of the K (five) tests. We applied scikit-learn 1.1.0 as ProTracker's backbone ML framework and all experiments are conducted on a multi-core server with an Intel Xeon Gold 6138 Skylake CPU with 40 cores, each running at 2.0 GHz, and 192 GB DDR4 memory.

Results. We first perform the bottom-level cross-validation on models with D_{ht} of six datasets. After tuning hyper-parameters, LR, KNN, MLP, and GBRT achieves 0.81, 0.95, 0.85, 0.95 averaged R^2 score on training and 0.77, 0.90, 0.81, 0.89 averaged R^2 score on testing, respectively. Then we perform the top-level cross-validation on tuned models with D_{ms}. With the hyper-parameters picked by the bottom-level cross-validation, LR, KNN, MLP, and GBRT achieves 0.80, 0.93, 0.83, 0.95 averaged R^2 score on training and 0.77, 0.88, 0.79, 0.88 averaged R^2 score on testing, respectively[2]. At last, based on the models picked by the top-level cross-validation for each dataset, we perform forecasts and achieve an averaged R^2 score of 0.86, as shown in Table 5.3.

The high accuracy benefits from (1) properly selected features, which have a high signal-to-noise ratio and are strongly related to the forecast target (programming progress); (2) we limit the feature numbers, to reduce the difficulty of tuning models and avoid the curse of dimensionality [13]; (3) though we have limited samples in each dataset, we apply cross-validations on tuning parameters and picking models to improve the robustness and avoid the overfitting.

From the model aspect, KNN and GBRT perform better than LR because they capture the non-linear signals from features, MLP has a similar ability, but it's difficult to tune MLP for its complex structure. From the dataset aspect, the sample size is not directly related to the accuracy. ProTracker achieves high accuracy even among datasets with a small sample size since the dataset is high quality (e.g. accurate labels, similar structure for each file, etc).

5.4.3 Overhead

Evaluation setup. We implement ProTracker's frontend based on VS Code extension APIs [105] and the extension runs on VS Code version 1.70.0. To evaluate the frontend's overhead, we randomly sample 50 homework files from each dataset for exper-

[2]Due to the limited space, the accuracy result of two-level cross-validation will be presented in the final version.

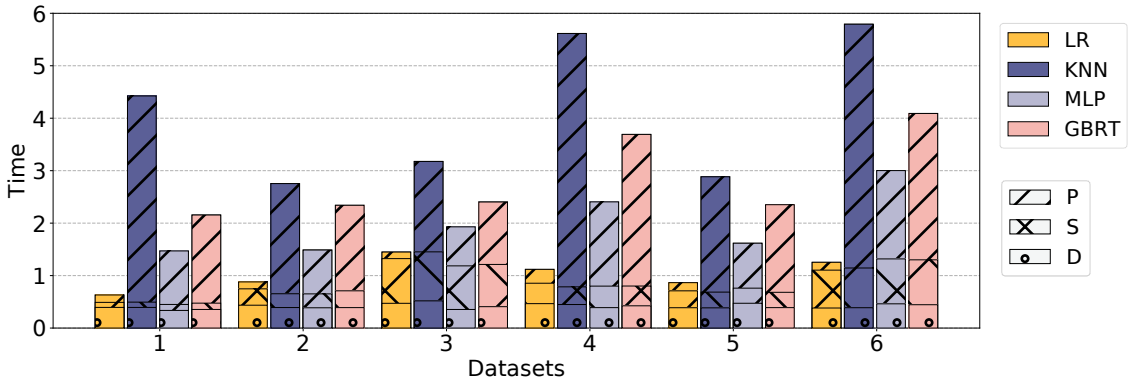


Figure 5.5: Overhead of ProTracker's frontend (run as a VS Code extension).

iments. To better reflect the overhead, for each file we perform a forecast on full code text (with 100% programming progress) because more code text leads to higher overhead in static analysis. We evaluate the frontend with all four candidate models regardless of the accuracy. We test 100 times for each file on each model, the results are presented as the average time of experiments. All experiments are conducted on a 2018 MackBook Pro with a 2.3 GHz Quad-Core Intel Core I5 and 16 GB DDR3 memory.

Results. Figure 5.5 shows the overhead of ProTracker's frontend. The overhead is decomposed into three parts: static analysis (feature extraction), model forecast, and driver, the averaged time is 0.46ms, 1.62ms, and 0.41ms, respectively. On average, it takes 2.5ms for each programming progress prediction process on PCs, which shows the overhead caused by ProTracker is neglectable. The driver overhead is not affected by the choice of models or datasets. The static analysis overhead is dependent on the code content, thus its overhead varies among different datasets. On average, LR, KNN, MLP, and GBRT take 0.16ms, 3.2ms, 1.12ms, and 1.97ms for forecasting, respectively. KNN and GBRT show superior performance at Section 5.4.2, but LR and MLP may be better solutions for programming progress prediction when the frontend runs on low-end hardware.

5.5 Limitations

First, ProTracker requires retraining the model when applying it to courses with languages other than Python. Fortunately, such retraining requires lightweight efforts, from our experiences. Second, while ProTracker shows high estimation accuracy, ProTracker may suffer from some inaccurate estimation in more advanced Python programming courses. Many existing ML techniques can solve this issue, we will employ them in the future. Third, ProTracker does not directly consider the code's correctness. We will integrate the testing and debugging efforts into our progress prediction model in the future.

5.6 Summary

This paper presents ProTracker, the first end-to-end solution to estimate the progress of programming assignments with machine learning techniques. It employs static analysis to extract features from assignment samples from previous semesters, and applies a two-level cross-validation method for tuning and selecting the proper machine-learning model. Its frontend runs as a Visual Studio Code extension and performs real-time programming progress estimation for students. Our evaluation demonstrates that ProTracker achieves an average R_2 score of 0.86 with 2.5ms overhead for real-time estimating programming progress of assignments.

Chapter 6

Conclusion and Future Research Directions

6.0.1 Summary of Dissertation Contributions

As an important research topic in the software engineering community, software inefficiency analysis is needed everywhere in computer systems ranging from smartphones to supercomputers. We detect and eliminate software inefficiency by pinpointing and optimizing useless and redundant memory operations. This dissertation demonstrates program analysis for both software engineers and students. We perform two works to detect program inefficiency and provide optimization guidance for software packages; two works to improve students' code quality and performance. By renovating the pedagogical learning environment and adding new techniques, we better understand and address the challenges that students have during programming, improve CS teaching and learning for undergraduate students, and further succeed in CS fundamental studies. We elaborate on the following conclusions:

Inefficiency Detection in Compiler Optimizations. We develop CIDetector and CIBench and leverage them to perform the first study on compiler-introduced and compiler-missed optimizations in fully optimized binary codes. We study five state-of-

the-art compilers, including three `gcc` compilers of different versions, and the most recent `icc` and `llvm` releases. This work is the first systematic study of compiler-related inefficiencies, it offers several insights that are valuable for scientific programmers, compiler writers, and tool developers, including inspiring programmers of compiler limitations, showing that different compilers have different optimization strategies, and motivating the necessity of analysis tools. This work is accepted to ICS'20.

Interaction Inefficiency Detection for Python Applications. We present PieProf, a lightweight profiler to pinpoint the interaction inefficiencies in Python applications. PieProf leverages the CL algorithm in a multi-language environment, in which Python is used to govern the semantics and native libraries are used to execute computation. PieProf identifies interaction inefficiencies with two common inefficiency patterns: redundant loads and redundant stores. PieProf works for production Python software packages in commodity CPU processors without modifying the software stacks. This work is accepted to ESEC/FSE'21.

Computer Science education research. We report the experience of integrating VS Code into a CS1 Python programming course, together with comprehensive guidance, it significantly balances the teaching resources and shortens the students' learning curves. In addition, we focus on the development of education-related VS Code extensions. We propose ProTracker, the first end-to-end solution to estimate the progress of programming assignments with machine learning techniques. ProTracker employs static analysis to extract features from assignment samples from previous semesters, and applied a two-level cross-validation method for tuning and selecting the proper machine-learning model. It runs as a VS Code extension and performs real-time programming progress estimation for students. These works are under-reviewing by CSE (CS education) conferences.

6.0.2 Future Directions

Performance profiling on emerging platforms and architectures. In our future direction, we will explore performance profiling on emerging platforms and architectures. Our current profiling tools run on x86 only, we plan to extend the inefficiency detection to the ARM architecture by leveraging DynamoRio [20]. Moreover, many popular Python functions and programs run on GPUs to achieve data parallelism, we plan to extend the interaction inefficiency detection to Python applications with GPU acceleration.

Education-related extensions. We plan to further expand the current advanced learning environment, integrated with more pedagogical approaches, to better improve the undergraduate/graduate students' learning experiences, including the development of extensions that further support education-related activities. Specifically, we are developing an extension that enhances students' visual comfort during programming activities, it provides students with recommendations for meticulously crafted code themes that integrate meaningful and intuitive visual representations, known as semantic coloring code themes, tailored to their programming environment better engage students.

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Laksono Adhianto, Sinchan Banerjee, Mike Fagan, Mark Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan R Tallent. Hpctoolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010.

[3] Abdulmalek Al-Gahmi, Yong Zhang, and Hugo Valle. Jupyter in the classroom: An experience report. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1*, pages 425–431, 2022.

[4] H Alaidaros and Mazni Omar. Software project management approaches for

monitoring work-in-progress: A review. *Journal of Engineering and Applied Sciences*, 12(15):3851–3857, 2017.

[5] Hamzah Alaidaros, Mazni Omar, Rohaida Romli, and Adnan Hussein. The development and evaluation of a progress monitoring prototype tool for software project management. In *2019 First International Conference of Intelligent Computing and Engineering (ICOICE)*, pages 1–9. IEEE, 2019.

[6] Glenn Ammons, Thomas Ball, and James R Larus. Exploiting hardware performance counters with flow and context sensitive profiling. *ACM Sigplan Notices*, 32(5):85–96, 1997.

[7] Matthew Arnold and Peter F Sweeney. *Approximating the calling context tree via sampling*. IBM TJ Watson Research Center Yorktown Heights, New York, US, 2000.

[8] Atom. Atom homepage. `https://atom.io/`, 2022.

[9] Gina R Bai, Justin Smith, and Kathryn T Stolee. How students unit test: Perceptions, practices, and pitfalls. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*, pages 248–254, 2021.

[10] Lorena A Barba, Lecia J Barker, Douglas S Blank, Jed Brown, Allen B Downey, Timothy George, Lindsey J Heagy, Kyle T Mandli, Jason K Moore, David Lippert, et al. Teaching and learning with jupyter. *Recuperado: https://jupyter4edu. github. io/jupyter-edu-book*, 2019.

[11] Elena García Barriocanal, Miguel-Ángel Sicilia Urbán, Ignacio Aedo Cuevas, and Paloma Díaz Pérez. An experience in integrating automated unit testing practices in an introductory programming course. *ACM SIGCSE Bulletin*, 34(4):125–128, 2002.

[12] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and David Binkley. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In *2012 28th IEEE international conference on software maintenance (ICSM)*, pages 56–65. IEEE, 2012.

[13] Richard Bellman. Dynamic programming. *Science*, 153(3731):34–37, 1966.

[14] Emery D Berger. Scalene: Scripting-language aware profiling for python. *arXiv preprint arXiv:2006.03879*, 2020.

[15] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX, pages 117–128, New York, NY, USA, 2000. ACM.

[16] Susan Bergin and Ronan Reilly. Predicting introductory programming performance: A multi-institutional multivariate study. *Computer Science Education*, 16(4):303–323, 2006.

[17] L Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, et al. An updated set of basic linear algebra subprograms (blas). *ACM Transactions on Mathematical Software*, 28(2):135–151, 2002.

[18] Gary D Boetticher, Wei Ding, Charles Moen, and Kwok-Bun Yue. Using a pre-assessment exam to construct an effective concept-based genetic program for predicting course success. *ACM SIGCSE Bulletin*, 37(1):500–504, 2005.

[19] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, and Skye Wanderman-Milne. JAX: composable transformations of Python+NumPy programs, 2018.

[20] Derek Bruening and Timothy Garnett. Building dynamic instrumentation tools with dynamorio. In *Proc. Int. Conf. IEEE/ACM Code Generation and Optimi zation (CGO), Shen Zhen, China*, 2013.

[21] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. Spec cpu2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, pages 41–42, 2018.

[22] Milind Chabbi, Xu Liu, and John Mellor-Crummey. Call paths for pin tools. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, page 76. ACM, 2014.

[23] Milind Chabbi and John Mellor-Crummey. DeadSpy: A Tool to Pinpoint Program Inefficiencies. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 124–134, New York, NY, USA, 2012. ACM.

[24] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE international symposium on workload characterization (IISWC)*, pages 44–54. IEEE, 2009.

[25] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794, 2016.

[26] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Tvm: end-to-end optimization stack for deep learning. *arXiv preprint arXiv:1802.04799*, 11:20, 2018.

[27] Yu Chen and Zhenming Liu. On efficient constructions of checkpoints. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020*, 2020.

[28] Yu Chen, Ivy B Peng, Zhen Peng, Xu Liu, and Bin Ren. Atmem: Adaptive data placement in graph applications on heterogeneous memories. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, pages 293–304, 2020.

[29] Ching-Yao Chuang. Ggnn: A pytorch implementation of gated graph sequence neural networks. https://github.com/chingyaoc/ggnn.pytorch, 2018.

[30] Eui-Young Chung, Luca Benini, and Giovanni De Micheli. Energy Efficient Source Code Transformation based on Value Profiling. In *PROC. INTERNATIONAL WORKSHOP ON COMPILERS AND OPERATING SYSTEMS FOR LOW POWER*, 2000.

[31] Intel Co. Pin 3.10. https://software.intel.com/sites/landingpage/pintool/docs/97971/Pin/html/, 2019.

[32] Visual Studio Code. User interface. https://code.visualstudio.com/docs/getstarted/userinterface, 2022.

[33] Coral Collaboration. Coral-2 benchmarks. https://asc.llnl.gov/coral-2-benchmarks/, December 2017.

[34] James S Collofello and Marla Hart. Monitoring team progress in a software engineering project class. In *FIE'99 Frontiers in Education. 29th Annual Frontiers in Education Conference. Designing the Future of Science and Engineering Education. Conference Proceedings (IEEE Cat. No. 99CH37011*, volume 1, pages 11B4–7. IEEE, 1999.

[35] CPython Community. C-extensions for python. https://cython.org/#documentation, 2007.

[36] Keith Cooper, Jason Eckhardt, and Ken Kennedy. Redundancy elimination revisited. In *Proceedings of the 17th International Conference on Parallel architectures and compilation techniques*, pages 12–21, 2008.

[37] Ian Stapleton Cordasco. Flake8: Your tool for style guide enforcement. https://flake8.pycqa.org/en/latest/, 2010.

[38] Intel Corp. Nehalem performance monitoring unit programming guide. https://software.intel.com/content/dam/develop/external/us/en/documents/30320-nehalem-pmu-programming-guide-core.pdf, 2010.

[39] Tyne Crow, Andrew Luxton-Reilly, and Burkhard Wuensche. Intelligent tutoring systems for programming education: a systematic review. In *Proceedings of the 20th Australasian Computing Education Conference*, pages 53–62, 2018.

[40] Vineesh Cutting and Nehemiah Stephen. A review on using python as a preferred programming language for beginners. 2021.

[41] Arnaldo Carvalho De Melo. The new linux'perf'tools. In *Slides from Linux Kongress*, volume 18, 2010.

[42] Steven J. Deitz, Bradford L. Chamberlain, and Lawrence Snyder. Eliminating Redundancies in Sum-of-product Array Computations. In *Proceedings of the 15th International Conference on Supercomputing*, ICS '01, pages 65–77, New York, NY, USA, 2001. ACM.

[43] Luca Della Toffola, Michael Pradel, and Thomas R. Gross. Performance problems you can fix: A dynamic analysis of memoization opportunities. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented*

Programming, Systems, Languages, and Applications, OOPSLA 2015, pages 607–622, New York, NY, USA, 2015. ACM.

[44] Luca Della Toffola, Michael Pradel, and Thomas R Gross. Performance problems you can fix: A dynamic analysis of memoization opportunities. *ACM SIGPLAN Notices*, 50(10):607–622, 2015.

[45] Roland DePratti. Using jupyter notebooks in a big data programming course. *Journal of Computing Sciences in Colleges*, 34(6):157–159, 2019.

[46] Monika Dhok and Murali Krishna Ramanathan. Directed test generation to detect loop inefficiencies. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 895–907, New York, NY, USA, 2016. ACM.

[47] Tom Dietterich. Overfitting and undercomputing in machine learning. *ACM computing surveys (CSUR)*, 27(3):326–327, 1995.

[48] Yufei Ding, Lin Ning, Hui Guan, and Xipeng Shen. Generalizations of the theory and deployment of triangular inequality for compiler-based strength reduction. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 33–48, New York, NY, USA, 2017. ACM.

[49] Yufei Ding and Xipeng Shen. Glore: Generalized loop redundancy elimination upon ler-notation. *Proc. ACM Program. Lang.*, 1(OOPSLA):74:1–74:28, October 2017.

[50] Paul J. Drongowski. Instruction-based sampling: A new performance analysis technique for amd family 10h processors. https://developer.amd.com/wordpress/media/2012/10/AMD_IBS_paper_EN.pdf, 2007.

[51] Tom Duff. Duff's device. https://www.lysator.liu.se/c/duffs-device.html, August 29 1988.

[52] John Edwards, Kaden Hart, and Christopher Warren. A practical model of student engagement while programming. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1*, pages 558–564, 2022.

[53] Robert G Edwards and Balint Joo. The chroma software system for lattice qcd. *arXiv preprint hep-lat/0409003*, 2004.

[54] Anthony Estey and Yvonne Coady. Can interaction patterns with supplemental study tools predict outcomes in cs1? In *Proceedings of the 2016 acm conference on innovation and technology in computer science education*, pages 236–241, 2016.

[55] Mary F. Fernández. Simple and Effective Link-time Optimization of Modula-3 Programs. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, PLDI '95, pages 103–115, New York, NY, USA, 1995. ACM.

[56] Ronald A Fisher. The use of multiple measurements in taxonomic problems. *Annals of eugenics*, 7(2):179–188, 1936.

[57] Eclipse Foundation. Eclipse homepage. https://www.eclipse.org/ide/, 2022.

[58] Python Software Foundation. cprofile. https://github.com/python/cpython/blob/master/Lib/cProfile.py, 2006.

[59] Python Software Foundation. Python document: Thread state and the global interpreter lock. https://docs.python.org/3/c-api/init.html#thread-state-and-the-global-interpreter-lock, 2010.

[60] Python Software Foundation. Python implementations. https://wiki.python.org/moin/PythonImplementations, 2014.

[61] Ben Frederickson. py-spy: Sampling profiler for python programs. https://github.com/benfred/py-spy, 2018.

[62] Ahmed Gad. Numpycnn: Implementing convolutional neural networks from scratch. https://github.com/ahmedfgad/NumPyCNN, 2018.

[63] gak. pycallgraph: Python call graph. https://github.com/gak/pycallgraph/, 2013.

[64] Github. bannsec. https://github.com/bannsec/pySym, 2018.

[65] Brian Gough. *GNU scientific library reference manual*. Network Theory Ltd., 2009.

[66] Harvard Intelligent Probabilistic Systems Group. Autograd: Efficiently computes derivatives of numpy code. https://github.com/HIPS/autograd, 2015.

[67] Hristina Gulabovska and Zoltán Porkoláb. Survey on static analysis tools of python programs. In *SQAMIA*, 2019.

[68] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. What bugs live in the cloud? a study of 3000+ issues in cloud systems. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 7:1–7:14, New York, NY, USA, 2014. ACM.

[69] Charles R Harris, K Jarrod Millman, Stéfan J van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, et al. Array programming with numpy. *Nature*, 585(7825):357–362, 2020.

[70] Simon Haykin. *Neural networks: a comprehensive foundation*. Prentice Hall PTR, 1994.

[71] John L Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[72] Sylvain Henry, Hugo Bolloré, and Emmanuel Oseret. Towards the Generalization of Value Profiling for High-Performance Application Optimization. http://sylvain-henry.info/home/files/papers/shenry_2015_vprof.pdf, 2015.

[73] Jian Huang, Moinuddin K. Qureshi, and Karsten Schwan. An evolutionary study of linux memory management for fun and profit. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '16, pages 465–478, Berkeley, CA, USA, 2016. USENIX Association.

[74] Wei Huang, Shougata Ghosh, Sivakumar Velusamy, Karthik Sankaranarayanan, Kevin Skadron, and Mircea R Stan. Hotspot: A compact thermal modeling methodology for early-stage vlsi design. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(5):501–513, 2006.

[75] Michael J Hull, Daniel Powell, and Ewan Klein. Infandango: automated grading for student programming. In *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*, pages 330–330, 2011.

[76] Robert Hundt, Easwaran Raman, Martin Thuresson, and Neil Vachharajani. MAO – An Extensible Micro-architectural Optimizer. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 1–10, Washington, DC, USA, 2011. IEEE Computer Society.

[77] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[78] Intel. Intel® distribution for python. https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/distribution-for-python.html, 2017.

[79] Intel. Intel c++ compiler 19.1 developer guide and reference. https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-introducing-the-intel-c-compiler, December 16 2019.

[80] GNU Compiler Collection (GCC) Internals. Link time optimization, 2019.

[81] M Irlbeck et al. Deconstructing dynamic symbolic execution. *Dependable Software Systems Engineering*, 40:26, 2015.

[82] JetBrains. Essential tools for software developers and teams. https://www.jetbrains.com/, 2022.

[83] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 77–88, New York, NY, USA, 2012. ACM.

[84] Justin Johnson. Pytorch-example: the fundamental concepts of pytorch through self-contained examples. https://github.com/jcjohnson/pytorch-examples, 2017.

[85] Mark Scott Johnson. Some requirements for architectural support of software debugging. *ACM SIGARCH Computer Architecture News*, 10(2):140–148, 1982.

[86] Teresa Johnson, Mehdi Amini, and Xinliang David Li. Thinlto: scalable and incremental lto. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 111–121. IEEE, 2017.

[87] Jython. Jpython homepage. https://www.jython.org/, 1999.

[88] Takahiro Kamio and Hidehiko Masahura. A Value Profiler for Assisting Object-Oriented Program Specialization. In *Proceedings of Workshop on New Approaches to Software Construction*, 2004.

[89] Fotis Kapotos. Fourier-transform. https://github.com/fotisk07/Fourier-Transform, 2018.

[90] Selina Khoirom, Moirangthem Sonia, Borishphia Laikhuram, Jaeson Laishram, and Tekcham Davidson Singh. Comparative analysis of python and java for beginners. *Int. Res. J. Eng. Technol*, 7(8):4384–4407, 2020.

[91] Sarvasv Kulpati. An implementation of linear regression from scratch in python. https://github.com/sarvasvkulpati/LinearRegression, 2018.

[92] Chris Lattner and Vikram 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*, page 75. IEEE Computer Society, 2004.

[93] Yepang Liu, Chang Xu, and Shing-Chi Cheung. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1013–1024, New York, NY, USA, 2014. ACM.

[94] Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. Learning efficient convolutional networks through network slimming. In *The IEEE International Conference on Computer Vision (ICCV)*, Oct 2017.

[95] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. A study of linux file system evolution. *Trans. Storage*, 10(1):3:1–3:32, January 2014.

[96] John L Luckner and Sandy K Bowen. Teachers' use and perceptions of progress monitoring. *American annals of the deaf*, 155(4):397–406, 2010.

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

[98] YuLong Luo and GuangMing Tan. Optimizing Stencil Code via Locality of Computation. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, pages 477–478, 2014.

[99] Shahin Mahdizadehaghdam, Ashkan Panahi, Hamid Krim, and Liyi Dai. Deep dictionary learning: A parametric network approach. *IEEE Transactions on Image Processing*, 28(10):4790–4802, 2019.

[100] Saeed Maleki, Yaoqing Gao, Maria J. Garzarán, Tommy Wong, and David A. Padua. An evaluation of vectorizing compilers. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 372–382, Washington, DC, USA, 2011. IEEE Computer Society.

[101] Samiha Marwan, Preya Shabrina, Alex Milliken, Ian Menezes, Veronica Catete, Thomas W Price, and Tiffany Barnes. Promoting students' progress-monitoring behavior during block-based programming. In *21st Koli Calling International Conference on Computing Education Research*, pages 1–10, 2021.

[102] Robert E McLear, DM Scheibelhut, and E Tammaru. Guidelines for creating a debuggable processor. *ACM SIGARCH Computer Architecture News*, 10(2):100–106, 1982.

[103] Rodrigo Pessoa Medeiros, Geber Lisboa Ramalho, and Taciana Pontual Falcão. A systematic literature review on teaching and learning introductory programming in higher education. *IEEE Transactions on Education*, 62(2):77–90, 2018.

[104] Microsoft. Intellisense. https://code.visualstudio.com/docs/editor/intellisense, 2022.

[105] Microsoft. Vs code: Extension guides. https://code.visualstudio.com/api/extension-guides/overview, 2022.

[106] Microsoft. What is visual studio live share. https://learn.microsoft.com/en-us/visualstudio/liveshare/, 2022.

[107] Mircosoft. Vs code marketplace. https://marketplace.visualstudio.com/, 2022.

[108] Matthias S Müller, John Baron, William C Brantley, Huiyu Feng, Daniel Hackenberg, Robert Henschel, Gabriele Jost, Daniel Molka, Chris Parrott, Joe Robichaux, et al. Spec omp2012—an application benchmark suite for parallel systems using openmp. In *International Workshop on OpenMP*, pages 223–236. Springer, 2012.

[109] Jonathan P Munson. Metrics for timely assessment of novice programmers. *Journal of Computing Sciences in Colleges*, 32(3):136–148, 2017.

[110] Robert Muth, Scott A. Watterson, and Saumya K. Debray. Code Specialization Based on Value Profiles. In *Proceedings of the 7th International Symposium on Static Analysis*, SAS '00, pages 340–359, London, UK, UK, 2000. Springer-Verlag.

[111] Thomas L Naps, Guido Rössling, Vicki Almstrum, Wanda Dann, Rudolf Fleischer, Chris Hundhausen, Ari Korhonen, Lauri Malmi, Myles Mc-

Nally, Susan Rodger, et al. Exploring the role of visualization and engagement in computer science education. In *Working group reports from ITiCSE on Innovation and technology in computer science education*, pages 131–152. 2002.

[112] Samy S Abu Naser. Developing an intelligent tutoring system for students learning to program in c++. *Information Technology Journal, Scialert*, 7(7):1055–1060, 2008.

[113] Khanh Nguyen and Guoqing Xu. Cachetor: Detecting cacheable data to remove bloat. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 268–278, 2013.

[114] Thieu Nguyen. Implement the-state-of-the-art meta-heuristic algorithms using python (numpy). `https://github.com/thieunguyen5991/metaheuristics`, 2019.

[115] Thieu Nguyen, Binh Minh Nguyen, and Giang Nguyen. Building resource auto-scaler with functional-link neural network and adaptive bacterial foraging optimization. In *International Conference on Theory and Applications of Models of Computation*, pages 501–517. Springer, 2019.

[116] Thieu Nguyen, Nhuan Tran, Binh Minh Nguyen, and Giang Nguyen. A resource usage prediction system using functional-link and genetic algorithm neural network for multivariate cloud metrics. In *2018 IEEE 11th conference on service-oriented computing and applications (SOCA)*, pages 49–56. IEEE, 2018.

[117] A. Nistor, L. Song, D. Marinov, and S. Lu. Toddler: Detecting performance problems via similar memory-access patterns. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 562–571, May 2013.

[118] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. Toddler: Detecting performance problems via similar memory-access patterns. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 562–571. IEEE, 2013.

[119] Numpy. Source code of `array_subscript` function. https://github.com/numpy/numpy/blob/5de64de6dbdf89b1bd8828c59393c4239364755a/numpy/core/src/multiarray/mapping.c#L1508, 2009.

[120] Taewook Oh, Hanjun Kim, Nick P. Johnson, Jae W. Lee, and David I. August. Practical Automatic Loop Specialization. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 419–430, New York, NY, USA, 2013. ACM.

[121] Travis E Oliphant. *A guide to NumPy*, volume 1. Trelgol Publishing USA, 2006.

[122] GNU Organization. The gnu compiler collection. https://gcc.gnu.org/, 2019.

[123] Rohan Padhye and Koushik Sen. Travioli: A dynamic analysis for detecting data-structure traversals. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 473–483, Piscataway, NJ, USA, 2017. IEEE Press.

[124] Darío López Padial. Technical analysis library in python. https://github.com/bukosabino/ta, 2018.

[125] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[126] Arnold Pears, Stephen Seidman, Lauri Malmi, Linda Mannila, Elizabeth Adams, Jens Bennedsen, Marie Devlin, and James Paterson. A survey of literature on the teaching of introductory programming. *Working group reports on ITiCSE on Innovation and technology in computer science education*, pages 204–223, 2007.

[127] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Van-

derplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[128] Jean-Philippe Pellet, Amaury Dame, and Gabriel Parriaux. How beginner-friendly is a programming language? a short analysis based on java and python examples. 2019.

[129] Vincent Pelletier. Pprofile: Line-granularity, thread-aware deterministic and statistic pure-python profiler. https://github.com/vpelletier/pprofile, 2013.

[130] Filipe Dwan Pereira, Elaine H. T. Oliveira, David Fernandes, and Alexandra Cristea. Early performance prediction for cs1 course students using a combination of machine learning and an evolutionary algorithm. In *2019 IEEE 19th International Conference on Advanced Learning Technologies (ICALT)*, volume 2161-377X, pages 183–184, 2019.

[131] Fernando Perez and Brian E Granger. Project jupyter: Computational narratives as the engine of collaborative data science. *Retrieved September*, 11(207):108, 2015.

[132] Leif E Peterson. K-nearest neighbor. *Scholarpedia*, 4(2):1883, 2009.

[133] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[134] pydata. Numexpr: Fast numerical expression evaluator for numpy. https://github.com/pydata/numexpr, 2009.

[135] PYPL. Popularity of programming language. http://pypl.github.io/PYPL.html, 2022.

[136] pythonprofilers. Memory profiler. https://github.com/pythonprofilers/memory_profiler/, 2012.

[137] PyTorch. Datasets, transforms and models specific to computer vision. https://github.com/pytorch/vision, 2017.

[138] Keith Quille and Susan Bergin. Programming: predicting student success early in cs1. a re-validation and replication study. In *Proceedings of the 23rd annual ACM conference on innovation and technology in computer science education*, pages 15–20, 2018.

[139] Mahesh Rajan, Douglas W. Doerfler, and Simon David Hammond. Trinity benchmarks on the intel xeon phi (knights corner). 1 2015.

[140] Payam Refaeilzadeh, Lei Tang, and Huan Liu. Cross-validation. *Encyclopedia of database systems*, 5:532–538, 2009.

[141] James Reinders. Vtune performance analyzer essentials. *Intel Press*, 2005.

[142] Charles Reis and Robert Cartwright. Taming a professional ide for the classroom. In *Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 156–160, 2004.

[143] Joe Rickerby. pyinstrument. https://github.com/joerick/pyinstrument, 2014.

[144] Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 12–27. ACM, 1988.

[145] Nancy Safer and Steve Fleischman. Research matters: How student progress monitoring improves instruction. *Educational Leadership*, 62(5):81–83, 2005.

[146] Norsaremah Salleh, Emilia Mendes, and John Grundy. Empirical studies of pair programming for cs/se teaching in higher education: A systematic literature review. *IEEE Transactions on Software Engineering*, 37(4):509–525, 2010.

[147] Savannah/kernel.org. The libunwind project homepage. https://www.nongnu.org/libunwind/, 2005.

[148] Carsten Schulte and Jens Bennedsen. What do teachers teach in introductory programming? In *Proceedings of the second international workshop on Computing education research*, pages 17–28, 2006.

[149] George AF Seber and Alan J Lee. *Linear regression analysis*. John Wiley & Sons, 2012.

[150] Marija Selakovic and Michael Pradel. Performance issues and optimizations in javascript: An empirical study. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 61–72, New York, NY, USA, 2016. ACM.

[151] Julian Seward. bzip2. http://www.bzip.org/, 2000.

[152] Ahmad Slim, Gregory L. Heileman, Jarred Kozlick, and Chaouki T. Abdallah. Predicting student success based on prior performance. In *2014 IEEE Symposium on Computational Intelligence and Data Mining (CIDM)*, pages 410–415, 2014.

[153] Linhai Song and Shan Lu. Performance diagnosis for inefficient loops. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 370–380, Piscataway, NJ, USA, 2017. IEEE Press.

[154] Linhai Song and Shan Lu. Performance diagnosis for inefficient loops. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 370–380. IEEE, 2017.

[155] Pamela J Springer, Patricia Johnson, Bonnie Lind, Eldon Walker, Joanne Clavelle, and Nancy Jensen. The idaho dedicated education unit model: cost-effective, high-quality education. *Nurse educator*, 37(6):262–267, 2012.

[156] KR Srinath. Python–the fastest growing programming language. *International Research Journal of Engineering and Technology*, 4(12):354–357, 2017.

[157] M Srinivas, B Sinharoy, RJ Eickemeyer, R Raghavan, S Kunkel, T Chen, W Maron, D Flemming, A Blanchard, P Seshadri, et al. Ibm power7 performance modeling, verification, and evaluation. *IBM Journal of Research and Development*, 55(3):4–1, 2011.

[158] stackoverflow. 2021 developer survey. https://insights.stackoverflow.com/survey/2021#section-most-popular-technologies-integrated-development-environmen 2021.

[159] Stasi. Numpywdl: Implement wide & deep algorithm by using numpy. https://github.com/stasi009/NumpyWDL, 2018.

[160] Daniel H Steinberg. The effect of unit tests on entry points, coupling and cohesion in an introductory java programming course. In *XP Universe*, volume 8. Citeseer, 2001.

[161] Lee Stott. Irisdata: Iris data example python numpy. https://github.com/leestott/IrisData, 2017.

[162] Pengfei Su, Qingsen Wang, Milind Chabbi, and Xu Liu. Pinpointing performance inefficiencies in java. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 818–829, 2019.

[163] Pengfei Su, Shasha Wen, Hailong Yang, Milind Chabbi, and Xu Liu. Redundant loads: A software inefficiency indicator. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, page 982–993. IEEE Press, 2019.

[164] Pengfei Su, Shasha Wen, Hailong Yang, Milind Chabbi, and Xu Liu. Redundant loads: A software inefficiency indicator. *arXiv preprint arXiv:1902.05462*, 2019.

[165] Jialiang Tan, Yu Chen, Zhenming Liu, Bin Ren, Shuaiwen Leon Song, Xipeng Shen, and Xu Liu. Toward efficient interactions between python and native libraries. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1117–1128, 2021.

[166] The PyPy Team. Pypy homepage. https://www.pypy.org/index.html, 2011.

[167] XLA Team et al. Xla-tensorflow compiled, 2017.

[168] Using the GNU Compiler Collection (GCC). Program instrumentation options. https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html, 2019.

[169] timothycrosley. Frosted documentation. https://pypi.org/project/frosted/, 2014.

[170] Daniel Toll, Tobias Olsson, Morgan Ericsson, and Anna Wingkvist. Fine-grained recording of student programming sessions to improve teaching and time estimations. In *International journal of engineering education*, volume 32, pages 1069–1077. Tempus Publications, 2016.

[171] Linda Torczon and Keith Cooper. *Engineering A Compiler*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2011.

[172] Gabriele N. Tornetta. Austin: A frame stack sampler for cpython. https://github.com/P403n1x87/austin, 2018.

[173] Python Tutor. Learn python, javascript, c, c++, and java. https://pythontutor.com/, 2021.

[174] Uber. Pyflame: A ptracing profiler for python. https://github.com/uber-archive/pyflame, 2016.

[175] Muhammad Usman, Ricardo Britto, Lars-Ola Damm, and Jürgen Börstler. Effort estimation in large-scale software development: An industrial case study. *Information and Software technology*, 99:21–40, 2018.

[176] Marat Valiev, Eric J Bylaska, Niranjan Govind, Karol Kowalski, Tjerk P Straatsma, Hubertus JJ Van Dam, Dunyou Wang, Jarek Nieplocha, Edoardo Apra, Theresa L Windus, et al. Nwchem: A comprehensive and scalable open-source solution for large scale molecular simulations. *Computer Physics Communications*, 181(9):1477–1489, 2010.

[177] Sander Valstar, Sophia Krause-Levy, Alexandra Macedo, William G Griswold, and Leo Porter. Faculty views on the goals of an undergraduate cs education and the academia-industry gap. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, pages 577–583, 2020.

[178] Sander Valstar, Caroline Sih, Sophia Krause-Levy, Leo Porter, and William G Griswold. A quantitative study of faculty views on the goals of an undergraduate cs program and preparing students for industry. In *Proceedings of the 2020 ACM Conference on International Computing Education Research*, pages 113–123, 2020.

[179] Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22, 2011.

[180] Eric Van Dusen. Jupyter for teaching data science. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, pages 1399–1399, 2020.

[181] Kelsey Van Haaster and Dianne Hagan. Teaching and learning with bluej: an evaluation of a pedagogical tool. *Issues in Informing Science & Information Technology*, 1, 2004.

[182] Kurt VanLehn. The relative effectiveness of human tutoring, intelligent tutoring systems, and other tutoring systems. *Educational psychologist*, 46(4):197–221, 2011.

[183] Gitte Vanwinckelen and Hendrik Blockeel. On estimating model accuracy with repeated cross-validation. In *BeneLearn 2012: Proceedings of the 21st Belgian-Dutch conference on machine learning*, pages 39–44, 2012.

[184] Arto Vihavainen. Predicting students' performance in an introductory programming course using data from students' own programming process. In *2013 IEEE 13th International Conference on Advanced Learning Technologies*, pages 498–499. IEEE, 2013.

[185] vim. Vim the editor. https://www.vim.org/, 2022.

[186] Jeffrey S Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)*, 11(1):37–57, 1985.

[187] Chao-Ming Wang. Numpydl: Numpy deep learning library. https://github.com/oujago/NumpyDL, 2017.

[188] Christopher Watson, Frederick WB Li, and Jamie L Godwin. Predicting performance in an introductory programming course by logging and analyzing student programming behavior. In *2013 IEEE 13th international conference on advanced learning technologies*, pages 319–323. IEEE, 2013.

[189] Mark N. Wegman and F. Kenneth Zadeck. Constant Propagation with Conditional Branches. *ACM Trans. Program. Lang. Syst.*, 13(2):181–210, Apr 1991.

[190] Shasha Wen, Milind Chabbi, and Xu Liu. Redspy: exploring value locality in software. In *ACM SIGARCH Computer Architecture News*, volume 45, pages 47–61. ACM, 2017.

[191] Shasha Wen, Xu Liu, John Byrne, and Milind Chabbi. Watching for software inefficiencies with witch. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 332–347, 2018.

[192] Shasha Wen, Xu Liu, and Milind Chabbi. Runtime Value Numbering: A Profiling Technique to Pinpoint Redundant Computations. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT)*, PACT '15, pages 254–265, Washington, DC, USA, 2015. IEEE Computer Society.

[193] Brenda Cantwell Wilson and Sharon Shrock. Contributing to success in an introductory computer science course: a study of twelve factors. *Acm sigcse bulletin*, 33(1):184–188, 2001.

[194] Jungsoon Yoo, Sung Yoo, Chris Lance, and Judy Hankins. Student progress monitoring tool using treeview. *ACM SIGCSE Bulletin*, 38(1):373–377, 2006.

[195] Victor Zhou. A convolution neural network (cnn) from scratch. `https://github.com/vzhou842/cnn-from-scratch`, 2019.

[196] Weijie Zhou, Yue Zhao, Guoqiang Zhang, and Xipeng Shen. Harp: Holistic analysis for refactoring python-based analytics programs. In *42nd International Conference on Software Engineering*, 2020.

[197] YiFei Zhou. Guppy 3: A python programming environment and heap analysis toolset. `https://github.com/zhuyifei1999/guppy3/`, 2019.

VITA

Jialiang Tan

Jialiang Tan is a Ph.D. candidate in the Department of Computer Science at William & Mary. She is advised by Dr. Xu Liu. Her research lies in high-performance computing, program/software analysis, and CS education research. Previously, she received her Bachelor of Engineering in Information Security from Sichuan University, China, in 2015. She received her Master of Science in Computer Science from Arizona State University in 2016, and her Master of Science in Computer Science from William & Mary in 2019.