View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Illinois Digital Environment for Access to Learning and Scholarship Repository

© 2021 Amarin Phaosawasdi

AUTOMATIC TEST GENERATION FOR THE DETECTION OF PERFORMANCE BUGS IN CODE OPTIMIZATION

BY

AMARIN PHAOSAWASDI

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2021

Urbana, Illinois

Doctoral Committee:

Professor David Padua, Chair
Professor Darko Marinov
Professor William D. Gropp
Professor Emeritus David J. Kuck, Intel
Professor Ponnuswamy Sadayappan, University of Utah
Dr. Saeed Maleki, Microsoft Research

ABSTRACT

Software is everywhere in our daily lives, and it is important that software behaves in ways it is expected to. Testing is a widely accepted method for improving software quality. Testing detects the presence of bugs through comparing the actual outcome to the expected outcome of a computation.

Testing for correctness is a well-studied problem. Testing for correctness compares the actual outcome of a computation against its expected output. Typically, the expected output of a computation is unambiguous, since computations in computer software typically have clear semantics defined by the programming language.

However, testing for performance is less studied. The expected outcome of a test may require context-knowledge not apparent in the test program itself. For example, by simply inspecting the code of a web server, one cannot determine what is the expected throughput. This makes performance testing for performance a challenging task.

Testing compilers adds another layer of complexity. For compilers, a correctness bug during compiler optimization may introduce a bug in the resulting binary, even though the bug was not present in the source code. Similarly, a performance bug during optimization may cause inconsistencies in the runtimes of equivalent programs, where equivalent programs are defined as programs with identical outcomes but whose sources may differ through semantic-preserving transformations. Performance bugs prevent compilers from producing efficient code when they have the ability to do so.

Many testing techniques have been proposed. Random testing is a powerful testing technique often associated with test generation. It allows a large testing space to be explored efficiently through sampling and is suitable for large and complex software with a large testing space, such as compilers.

Random test generation for compilers have been shown to be effective in detecting correctness bugs. However, to the best of our knowledge, there is no previous study on random test generation for performance bugs in compilers. We believe one of the main reasons is the context-dependent nature when quantifying performance headroom.

We propose a random test generation infrastructure for evaluating the performance of compilers. We quantify the performance headroom of tests by borrowing existing ideas from previous studies. Namely, when a set of equivalent programs is compiled by a compiler, all programs should aim to perform as well as the best-performing program. Additionally, when a program is compiled by a set of compilers, all compilers should aim to generate code that

performs as well as the code generated by the best-performing compiler. We define metrics to evaluate compilers based on these ideas.

We used our system to evaluate four modern compilers – Intel's ICC, GNU's GCC, the Portland Group Inc.'s PGI compiler, and Clang – on how well they handle loop unrolling, loop interchange, and loop unroll-and-jam. Results suggest that ICC typically performs better than the other three compilers. On the other hand, our system also identified extreme outliers for ICC where, for example, one program becomes x180000 slower after unrolling a loop.

Due to the nature of random testing, we also study the methodologies required to achieve reproducible results by using statistical methods. We apply these methodologies to our compiler evaluation and provide evidence that our experiments are reproducible across different randomly generated collections of code segments.

Dedicated to my family.

ACKNOWLEDGMENTS

Throughout the seven and a half years in the Ph.D. program at the University of Illinois at Urbana-Champaign (UIUC), I have been fortunate enough to be surrounded by loving and remarkable people that made this journey a memorable and life-changing one.

First and foremost, thank you my advisor, Professor David Padua, who has always been a source of wisdom and optimism throughout my Ph.D. Several of our projects did not work out, but I always felt safe to fail, learn from it, and try again because of your encouragement. You have taught me a lot how to ask important questions and find answers to them. Thank you for giving me the freedom in choosing whatever topic interested me and for always making my success your top priority. I feel so fortunate to have you as my advisor.

Next, I would like to thank my doctoral committee members Professor Darko Marinov, Professor William Gropp, Professor David Kuck, Professor Ponnuswamy Sadayappan, and Saeed Maleki for your time and feedback on this work. Thank you for improving the writing, the experiment design, the literature study, and pointing out many aspects that I had not thought about that would make this work more applicable and well-received by other researchers.

Thank you Professor Grigore Roşu for convincing me to join UIUC. It was definitely a hard decision to make, but retrospectively it was the best decision. I joined the program with an interest in programming languages, but the department has allowed me to meet numerous faculty members not only in programming languages, but also in related fields such as formal methods, software engineering, compilers, and high-performance computing. It has broaden my research interests, which has made me become a better researcher today.

Thank you Professor Tao Xie for welcoming me to UIUC and answering all the questions I had as a first year student, especially about finding an advisor. Thank you for letting me sit in your group meetings to learn your advising style while also encouraging me to see other professors.

Thank you Professor Darko Marinov for being a mentor throughout my Ph.D. You taught me so much when I was your teaching assistant during my first semester here, and you have continued to be somebody I can always reach out to talk about anything. Your sense of humor is always appreciated.

Thank you Professor Jeremy Johnson, my first research collaborator. We worked together briefly on using Isabelle to prove compilation correctness and evaluating CompCert. It was a great experience that kicked start my Ph.D.

Thank you Professor Elsa Gunter for increasing my interest in formal methods. I enjoy stopping by your office and talking to you both about research and personal matters. I always have fun seeing you lead the formal methods seminar. I love your teaching style, and you inspire me to care deeply about students.

Thank you Peter Dinges, my Ph.D. ambassador. You have always been one of the main people from whom I seek advice during each pivotal moment in my Ph.D. career. You have inspired me to pay it forward and help other Ph.D. students. I have volunteered to be an ambassador every year that I had the chance.

Thank you Owolabi Legunsen and Sandeep Dasgupta, my Ph.D. cohorts. We have faced many of the same challenges together, and through those difficult times, we grew stronger together. Thank you for your help through every milestone, including practice presentations and examinations. Thank you for the random chats in the hallway and for stopping by my office. I am glad we met, and I hope this is a lifelong friendship.

Thank you Mohsen Vakilian for the opportunity to help with your research which eventually became my first conference paper. I learned a lot working alongside you. You also gave me the opportunity to present the paper on your behalf, while you had personal matters to handle. I felt unprepared, but it has pushed me to overcome my limits. In the end, it was an excellent experience that made me a better presenter and communicator. For this work, I would also like to thank Professor Michael Ernst, Professor Ralph Johnson, and Milos Gligoric for your help improving the presentation.

I interned at Huawei (USA) for 1.5 years, during which I had the opportunity to work alongside the best. Thank you Christopher Rodrigues for your guidance, especially with paper writing. We wrote two papers together, and I thoroughly enjoyed learning from you. Also, thank you for teaching me Haskell and how to think like a functional programmer. Thank you Peng Wu for teaching me how to navigate and survive a high-pressure working environment. I have also learned a tremendous amount on effective project planning and execution from you. It was also a pleasure interacting with the rest of the team including Salem Derisavi, Balajee Gurumoorthy, Xuejun Yang, Wilson Feng, Ganesh Bikshandi, Long Chen, Brice Dobry, Xiurong Zhu, Haichuan Wang, Ahmed Hussein, and Wooyoung Kim.

Thank you Abdulrahman Mahmoud and Muhammad Huzaifa for your help with all the practice presentations I have had here. Outside of research, I would also like to thank you for inviting me to join your soccer games. It was fun playing together. I hope we get to play again.

Thank you my labmates and colleagues in the neighboring labs including Thiago Teixeira, Sweta Pothukuchi, Justin Szaday, Tarun Prabhu, August Shi, Farah Hariri, Alex Gyori, Wing Lam, Lamyaa Eloussi, Wei Yang, Angello Astorga, and Xusheng Xiao for all the

fruitful conversations we have had.

Because of the Thai Student Association at UIUC, I have made many new friends over the years, many of which I am glad to call my UIUC family. These people have been crucial to my well-being of life outside of work. There are so many people to list, so please forgive me if I missed anyone. I would like to thank Sun, Song, Mick, Auy, Ice, March, North, Nam (Pitchaya), Tent, Ing, Oak, Poupée, Poom, Time, JR, Titi, Willy, Pingping, Music, Paii, Ford, Mye, Junior, Pong, Earn, Boat, Val, Pitcha, Parp, Au (Phakpoom), New, Near, First, Nat, Tee, CC, Title, Ume, Nam (Nicha), Pink, Teaw, Rakoon, Nop, Gath, Wave, Quinn, Tae (Tanitpong), Kwin, Ping, Park, Chanon, Koi, Micky, Team, Som, May, Ming, Oat, Tum, Shafeen, Plern, Panda, Meena, Win, Nam (Supicha), Tae (Naran), Blink, Wut, Fluke, Pui (Taya), Gloy, Champ, Bom (Nattapon), Yo, and anyone I may have missed for all the movie nights, video gaming sessions, music jamming, meals together, beers together, cooking sessions, soccer matches, and deep conversations we have had. Thinking back on those memories makes me smile every time.

Special thanks to North, Nam, Ing, and Poupée, my remaining close friends on campus and nearby, who made time to meet in person and spend time together during the most difficult times after the COVID-19 pandemic started. Also special thanks to Mick, Tent, Angel, Natty, Ford, JR, Junior, Music, Mye, Paii, Pingping, Poom, Time, Titi, and Willy, who have constantly kept me company virtually from far away after the pandemic started.

Thank you all of the CS academic staff, including but not limited to, Kathy Runck, Viveka Kudaligama, Elaine Wilson, Sherry Unkraut, Kimberly Bogle, Salome Liebenberg, and Maggie Metzger Chappell for helping me navigate through the complicated official processes in graduate school.

Finally, I cannot thank my family enough. Thank you dad for always encouraging me to study further. Without you, I would have not pursued a Ph.D. Thank you mom for your unconditional love and support, no matter what path I choose in life. I know Ph.D. took a while, but now I have finally graduated. To my soulmate, Dear, for supporting my decision to pursue a Ph.D. even if that meant we had to live separately for a many years. Thank you for all the sacrifices you have made for us. You have always put my needs above yours. I love you. Thank you Annie, my sister, for celebrating each and every milestone of my Ph.D. You are a great source of encouragement. Finally, thank you the rest of my extended family for the continuous support. I feel comfort knowing that I have an army of people rooting for my achievements.

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION 1
1.1 The Problem 1
1.2 Thesis Objective 3
1.3 Challenges in Testing for Performance 4
1.4 Challenges in Random Testing 5
1.5 Contributions 6
1.6 Thesis Organization 6

CHAPTER 2 RELATED WORK 8
2.1 Compiler Evaluation 8
2.2 Formal Method Approaches 9
2.3 Testing Approaches 10
2.4 Handling Randomness 12
2.5 Program Generation 13

CHAPTER 3 PERFORMANCE TEST GENERATION 14
3.1 Approach 14
3.2 Design Goals 15
3.3 System Overview 16
3.4 Random Test Generation 18
3.5 Pattern Generation 18
3.6 Fixed Patterns 22
3.7 Instance Generation 22
3.8 Mutation Generation 24
3.9 C Code Generation 27
3.10 Building and Running Executables 27
3.11 Reporting Results 28

CHAPTER 4 EVALUATION 29
4.1 Experiment Setup 29
4.2 Metrics 35
4.3 Statistical Quantification of Results 44
4.4 Compiler Evaluation: Loop Unroll 51
4.5 On the Space of Possible Patterns 58
4.6 Interpreting the Results 59
4.7 Compiler Evaluation: Loop Interchange 59
4.8 Compiler Evaluation: Loop Unroll and Jam 65
4.9 Reproducibility of Experiments 70

CHAPTER 5 THE REPRESENTATIVENESS OF SYNTHETIC PROGRAMS . . 71
5.1 Experiment Setup 71
5.2 Comparing Two Populations 73
5.3 Runtime Stability 75
5.4 Peer Speedup 75

CHAPTER 6 TESTING FOR CORRECTNESS 81

CHAPTER 7 CONCLUSIONS 83

REFERENCES 85

CHAPTER 1: INTRODUCTION

1.1 THE PROBLEM

Software has become ubiquitous. We use software systems in our daily lives. There are software systems for many domains including, but are not limited to, commerce, transportation, healthcare, finance, engineering, and science.

For software systems to be useful, we expect it to do the right thing. Since the properties of interest differ across application domains, it is important to be clear what doing the right thing means in each context. For example, airplane operating systems should not crash and should have low response time since not meeting these goals could cost lives. Security systems should not allow unauthorized access. A simulation in physics may not need highly accurate results, but it may be important to run the simulation reasonably fast, given the immense number of operations typically involved.

While programmers may put tremendous amounts of effort into writing software that meets the required properties, their effort may be diminished if the compiler that is used to build the software introduces incorrect behavior. Therefore, it is also important to ensure that compilers generate correct executables from correct source programs.

Compilers have made significant improvements over the decades. There has been numerous studies on improving the correctness of compilers. Compilers are correct when they do not introduce correctness bugs in the software during compilation. Safety-critical applications and security systems focus on preventing correctness bugs and benefit from research in the area of compiler correctness.

There has been tremendous effort in increasing the correctness of compilers. One area of current research is formal methods. Verified compilers, such as CompCert [1], are formally (and mechanically) proven to maintain a consistent behavior between source and target code. Another active area of research is testing. Testing is a less expensive alternative, complementing formal methods. Random test generation tools, such as CSmith [2], are shown to be effective in finding compiler bugs.

In the area of high performance computing, we are interested not only in correctness but also performance. For a fixed algorithm or problem, such as matrix multiplication, programmers have the choice of using highly-tuned libraries, such as the Intel Math Kernel Library [3] or code generation tools, such as the Tensor Contraction Engine [4] to gain high performance from the underlying hardware on which the code is run.

However, for any other arbitrary program, programmers have to rely on compilers to

do their best to provide high-performance code. Optimizing compilers carry out various transformations to increase the performance of programs [5]. The compiler needs to choose among the space of these transformations to produce the most efficient target code. Programs that are transformed by compiler transformations are semantically equivalent (or simply equivalent) programs since they produce the same result for each input data set.

Equivalent programs should have equivalent performance. The reason is that the compiler should be able to both apply and undo any transformation to these equivalent programs and automatically transform one into another. However, a recent study shows that modern compilers are **unstable**, generating binaries that differ widely in performance across equivalent programs [6]. When the performance of these programs differs widely, the developer must analyze the code and possibly massage it to obtain the best performing source program. Compiler instability means that the programmer does not know when the code is suboptimal and thus is one cause of performance bugs.

There are few previous studies related to performance bugs in compilers. These works have produced influential benchmarks. The Test Suite for Vectorizing Compilers (TSVC) [7] is a set of Fortran loops aimed at evaluating compiler effectiveness in vectorizing loops. The test suite was originally designed to evaluate vectorization capabilities of compilers for vector supercomputers. Later, the test suite was converted to C and was used to study compilers targeting microprocessor vector extensions [8]. The C version of TSVC contains 151 synthetic loops. It is designed to test the ability of compilers to apply important analyses and transformations, including but not limited to loop interchange, loop peeling, loop distribution, loop reversal, loop re-rolling, dependence analysis, induction variables elimination, scalar expansion, and identification of recurrences and reduction patterns. One of the main strengths of TSVC is its ability to evaluate the ability of compilers to apply different vectorization-enabling transformations. When a TSVC test performs poorly, compiler developers have a direction on what components need improvement. On the other hand, we consider the number of test cases in TSVC small to make a good evaluation of a large and complex system like a compiler. It is possible to write a compiler that does well only on the tests in TSVC and poorly on anything else.

The LORE loop repository [9] is a much larger collection of tests. The repository includes approximately 2500 loops extracted from many benchmarks, including TSVC. Not only does the loop repository include the original loops, the repository also includes their **mutations**, equivalent programs resulting from semantic-preserving code transformations. Mutations in the repository are created through loop interchange, tiling, unrolling, unroll-and-jam, and distribution. The loop transformations are applied when they are legal to do so and do not change the result of the output. When considering the mutations, the repository

contains almost 100,000 tests. An important question that arises is then how one should use these loops to evaluate compilers. While these programs are a collection of loops that represent real-world loops, each of the benchmarks represent loops from different domains. Merging all the benchmarks into a single collection or repository may not represent the domain of programs that a compiler evaluator is interested in, defeating the purpose of using real-world loops. Additionally, any follow up work using these loops would still need to determine whether the number of loops used for testing are sufficient or excessive to make any conclusions about the performance of the compiled programs or not.

The main strength of TSVC is the carefully designed tests to evaluate different transformation capabilities of compilers. The main strength of LORE is the large number of tests and the idea of creating mutations from original programs. In this thesis, we propose a strategy that combines the strength of the two studies by automatically generating a large number of tests that focus on testing compiler transformations. We further generalize the idea of generating tests into creating an infrastructure for evaluating the performance of compilers.

When generating tests, the space of all possible programs is too large to systematically enumerate and run. Sampling the space through random testing is an approach that is appropriate for this situation. It has been shown that random testing is effective in detecting correctness bugs in compilers [2]. We believe that it is also effective in detecting performance bugs in compilers.

We propose using a test generation infrastructure to evaluate the performance of compilers based on random testing. Testing large and complex systems like compilers require a large number of tests. Through test generation, we can create a large number of tests. Additionally, using semantic-preserving source level transformations to generate equivalent programs for testing, our classes of tests are naturally centered around compiler transformations. This gives users more insight of the types of transformations that may be problematic.

1.2 THESIS OBJECTIVE

This dissertation focuses on the study of the use of random test generation to evaluate the effectiveness of compilers in generating high performance code. We have developed an infrastructure for randomly generating tests. We show in our studies how the system can be used to evaluate compilers. We define metrics to quantify the effectiveness of compilers when compiling for performance. We discuss the uneven quality we observe across four modern compilers through our experiments. We also discuss the high reproducibility of our results. We quantify how well the generated tests represent real-world programs. Finally, we discuss

potential uses of the system for testing compiler correctness.

1.3 CHALLENGES IN TESTING FOR PERFORMANCE

The presence of a bug can be determined through testing by comparing the expected result to the actual result of a test. When the actual and expected results do not match, there is a bug. Therefore the main requirement for testing is to precisely define the actual and expected results of a program.

Correctness bugs have a clear definition. A program carries out a computation consisting of a series of operations. The computation produces an output. The expected output can be precisely determined, since the operations are typically defined mathematically by the programming language semantics. Problems may arise when dealing with floating point numbers and their round-off errors, but this problem is well-studied and it is possible to bound the round-off errors when checking for correctness [10].

Performance bugs on the other hand do not have a clear definition. One reason is that performance can be measured in many ways, such as overall runtimes, using the content of hardware counters, or measuring resource usage. Another reason, which is more elusive, is the difficulties in quantifying performance headroom. Possibilities include comparison of measured FLOPS with theoretical FLOPS, or comparison to the roofline in the roofline model. However, unlike expected correctness results, none of these measurements can be determined just by looking at the code.

Our work focuses on compiler bugs. Compilers may introduce bugs into programs during compilation. This means that even for a program whose source code has been proven to be correct, it is still possible for a buggy compiler to cause the resulting binary to be buggy. Compiler correctness bugs happen when the target program has a bug because of the compiler.

Similarly, equivalent programs should have approximately equivalent runtimes when compiled. However, compiler performance bugs can introduce inconsistencies in these runtimes and cause the compiler to produce less efficient code than what the compiler actually has the ability to produce.

In our work we use the runtime of a program as the measurement of performance. Alternatively, we could have used energy consumed or memory usage, but runtime is the typical focus in high-performance computing. The performance headroom of a program can be approximately quantified in two different ways.

First, the performance headroom can be approximated by studying equivalent programs compiled by the same compiler. This idea is based on insights observed by Gong et al [6].

The main idea is that for a given program, if it can be automatically transformed at the source-level by a compiler, the compiler should be able to reverse that transformation as well. An ideal compiler would then be able to convert back and forth and pick the best among the different versions of the same program. This property is called compiler stability. Given a set of equivalent programs, when a compiler compiles them, we know that the compiler is capable of achieving programs that are as fast as the fastest program in that set. We use that as the reference point and an approximation of the performance headroom of a program resulting from a transformation or a collection of transformations. We define metrics based on this idea. These stability metrics include the runtime stability (Section 4.2.3), vector speedup stability (Section 4.2.4), and cost model stability (Section 4.2.5).

And second, the performance headroom can also be approximated by measuring the outcome of having multiple compilers compile the same program. The fastest binary among the set of binaries produced by all compilers for the same program can be used as the reference runtime. The best runtime in this case may not be achievable across compilers, since different compilers may have completely different code generation schemes. However, we believe that a compiler developer would benefit from knowing how their compiler performs compared to others. Not only does it benefit the compiler developer, but it also benefits the compiler user who may be choosing the best compiler for the job among different compilers. We define metrics based on relative performance between peer compiler implementations. These peer comparison metrics include the top rank proportion (Section 4.2.7), bottom rank proportion (Section 4.2.8), better rank proportion (Section 4.2.9), and peer speedup (Section 4.2.10).

Another less severe, but noteworthy challenge in performance testing is that the test results can be affected easily by the environment in which the tests are conducted. We argue that it is easier to parallelize correctness tests than performance tests. This matters when testing time is in the magnitude of hours, as in running thousands of tests to evaluate compilers. In correctness testing, independent tests can be run in parallel. In performance testing, even though tests are independent, running tests in parallel within a shared memory system may affect the runtimes unless we ensure the testing environment for each test is isolated and identical. For each of our studies, we run tests sequentially. We limit the number of tests to balance between testing time and results that are reproducible, a problem we discuss in more detail in Section 4.3.

1.4 CHALLENGES IN RANDOM TESTING

One of the main strengths of random testing is that it allows us to statistically quantify results [11]. However, it also increases the burden on us to satisfy the data and sampling

requirements of any statistical methods we use. For systematic testing, there is no sampling, and this problem does not exist. Existing research shows that many studies do not rigorously use statistical methods to deal with randomness [12, 13].

It is important to satisfy any data requirements for the statistical methods being used. Otherwise, results may be invalid. For example, many statistical methods assume that the population from which samples are extracted has a normal distribution. In our studies, we show how we check this assumption (Section 4.3.4).

Random testing samples tests from a larger test population. Not only is it important to ensure any data requirements are met for statistical methods, it is equally important to be able to precisely describe the population and sampling methods. This is especially important for reproducibility of experiments, since the reproducer would need to sample data from the same population in the same way to attain the same results we report. In our studies, we carefully describe the random seed we use, the population we sample from, and how we sample data (Section 3.4).

1.5 CONTRIBUTIONS

Our main contributions are as follows.

- An infrastructure for generating programs and evaluating compilers that is easily extendable.
- Methodologies for identifying performance bugs in compilers through random testing and how to ensure reproducible results in the context of random testing.
- An evaluation of four modern compilers on how well they handle loop unroll, loop interchange, and loop unroll-and-jam, using our infrastructure.
- An evaluation on the reproducibility of our experiments.
- An evaluation on the representativeness of synthetic programs compared to real-world programs.

1.6 THESIS ORGANIZATION

The rest of this writing is organized as follows. Chapter 2 discusses related work. Chapter 3 describes our infrastructure for evaluating compilers. Chapter 4 details our methodology and evaluation results. Chapter 5 discusses the representativeness of synthetic programs.

Chapter 6 discusses how our infrastructure may be used for correctness testing. Finally, Chapter 7 concludes.

CHAPTER 2: RELATED WORK

In this chapter, we discuss studies related to our work. First, we discuss work directly related to compiler evaluation. Next, we discuss approaches to improving the quality of compilers, which include formal method approaches and testing approaches. Next, since randomness plays an important role in our approach, we discuss related studies on random testing and fuzzers. Finally, we discuss related work on program generation.

2.1 COMPILER EVALUATION

There is a great body of research on evaluating compilers. The original TSVC benchmark [7] in FORTRAN was a precursor to following works for evaluating vectorizing compilers. Later, the TSVC loops were converted into C, and several C-specific tests were also added so that they could specifically test vectorization capabilities of C compilers [14]. More recently, a similar study was conducted, but with more in depth analysis, including the effect of manual transformations on vectorizability [8]. While these studies provide useful insight to compiler developers, the limited number of tests may decrease the validity and generality of their results. Moreover, their tests focus on vectorization capabilities, while other optimizations done by the compiler may also have a large impact on performance. We believe that, in order to evaluate large and complex systems like compilers, a larger number of tests are needed, and we should evaluate compilers on different types of transformations.

Others have also studied the effect of source-level transformation on the compiled programs' performance. In 1971, Knuth performed a study of FORTRAN programs and how manually translating the these programs would improve performance [15]. Their work scored the performance of programs by hand by giving weights to different types of instructions. The nature of the study was an in-depth analysis of how optimizations may affect program performance. Our goal, on the other hand, is to provide an overview of compiler performance for a given population of programs for a set of comparable compilers. While we include some in-depth analysis on interesting cases, the large scope of the study prevents us from doing manual analysis of the quality of the code generated.

The LORE loop repository [9], along with an experiment to study compiler stability [6], provides a great groundwork for evaluating compilers when faced with equivalent versions of the source code. Their work includes various benchmarks gathered from multiple domains. Our work extends upon these experiments by approaching the problem through an extendable test infrastructure that uses a random test generation approach.

Another previous study that aligns with out goal is the study of the performance of the Ada compiler by Bassman [16]. The strength of their study lies in the approach of using different types of test cases. For example, test cases based on representative applications are used to evaluate the overall performance of compilers, while specific test cases are designed to test the optimization passes. Furthermore, they also design tests to evaluate code generation quality by assessing the performance of the binaries against hand-written assembly code. Our work builds upon the idea of having tests that both show overall performance of compilers while the tests are categorized by source-level transformation passes. A limitation of our work, however, is that we do not further differentiate performance inconsistencies whether they are due to compiler transformations or code generation. Doing so in the same manner as Bassman's study would require optimal assembly code for all the tests we generate.

Several tools, such as Bugfind [17] and vpoiso [18], have been developed to help pin-point buggy compiler optimization phases. These studies compare the results of a program when unoptimized against different optimization passes. The first optimization in a series of optimizations that cause the result to differ is the problematic optimizing phase. This process is done automatically to help compiler writers save time in debugging correctness errors in compilers. The testing techniques, comparing unoptimized code to the optimized version, is similar to our approach and other previous studies. These studies are useful, especially when large series of transformations is applied to a program. In our work, we study transformations in isolation, and our focus is on performance. We have not encountered the problems of pin-pointing problematic optimizations at this time.

Finally, tools such as the Notice framework [19] help compiler users find sequence of transformations that results in the best performance through an auto-tuning approach. Not only do they consider execution time as a measurement of performance, they also focus on the relationship between execution time and resource consumption when finding solutions. On the other hand, our work is only focused on reporting the overall performance of compilers and finding bugs. Their work informs us of possible future directions, in particular, considering more fine-grained optimization levels, considering resource usage, and providing solutions to end-users.

2.2 FORMAL METHOD APPROACHES

Researchers have proposed various approaches to make compilers less buggy through formal methods.

Aproaches such as a formally verified compiler [1] and translation validation [20], provide mechanical proofs certifying that the compiler does not introduce new bugs and the output

program from one pass does not change from the previous, respectively.

We are interested in performance bugs, and reasoning formally about "performance correctness" is not as straightforward. There is ongoing research on formally reasoning about resource usage in C programs [21]. However, the analysis only considers worst-case bounds and the analysis is at the C level. To analyze the performance of binaries, reasoning at the assembly level is required. Formal semantics at the assembly level exist, including MIPS [22] and X86 [23]. However, to the best of our knowledge, no work exists in reasoning about performance formally at the assembly level. One also needs to take into account the complexity of memory systems in modern hardware. With the current state of art, we believe that testing is a better alternative in evaluating performance of compilers.

2.3 TESTING APPROACHES

Random testing (or fuzzing) is a black-box testing approach known for a long time [24]. More recent works include white-box fuzzers that make use of source code knowledge to generate inputs [25, 26, 27]. While these works have been useful in the security domain, more recent works show that fuzzers can be used effectively in testing compilers [2, 28, 29]. We are not aware of studies based on random testing for performance in compilers. It has been effective in testing for correctness, and we show through our work that it is effective in testing for performance as well.

An important problem related to testing is test-case reduction. It is easier for developers to debug code when presented with smaller tests that exhibit bugs. Regehr et al. have proposed a test-case reduction system for C compiler bugs [30]. The system starts from an existing test-case that causes a bug. It then uses a technique called delta-debugging [31] that iteratively transforms the test to a smaller one that still causes the bug. Since our studies include samples drawn from a population of relatively small programs, we have not encountered this problem. Additionally, our main results include an overall evaluation of the performance of compilers. Test-case reduction would be useful when helping compiler writers debug specific outliers, such as tests generated from unrolling a loop multiple times.

When working with a large number of tests, a problem that may arise is the large amount of redundant test cases that trigger the same bug. Chen et al. called this the fuzzer taming problem and provided a solution to it [32]. Their solution associates each test with information, such as functions that are triggered in the compiler or the error message emitted by the compiler, then use a distance function to find a diverse set of tests that are not too close to each other. Again, our work mainly focuses on giving compiler writers an overview of the compiler's performance. When redundant bugs are triggered by different codes, it con-

tributes to the unstability of the compiler, and we do not want to discard this information. On the other hand, for compiler writers who are interested in the outliers we produce, our current approach is to simply report the N (defaulting to 10) lowest performing tests, which can contain duplicates. Adopting the approach from their work, would benefit us for this use case.

An alternative approach when dealing with small programs is systematic enumeration of all programs up to a certain size. One can do so by viewing programs more abstractively as skeletal programs with holes that will be filled by variable names later on, as described in Zhang's study [33]. A program is enumerated when we fill the holes with variables. This scheme includes redundancy in the enumerated programs, for example, when two variables are swapped for all occurences in the skeleton. Their approach reduces the problem to a set partitioning problem to remove these redundancies, reducing the total space of programs tremendously. The way we generate programs in our work, as described in Section 3.4 is similar to program skeletons. However, we sidestep the problem of exhaustive enumeration through random testing. Testing for performance bugs require more than program enumeration, namely also generating different problem sizes for the same program as well as semantic-equivalent set of programs. Even for small programs, the space becomes impractical to exhaustively enumerate. The combination of random testing and statistical methods allow us to make valid general claims about the compilers in our evaluation, while at the same time being able to generate corner-cases that detect possible performance issues.

Another approach to enumerating programs is through a technique called imperative generation [34]. Rather than using the grammar of the input to guide the generation, an approach used by CSmith [2], ASTGen [34] uses a framework based on imperative programming through an iterator interface and lazy evaluation instead. In their studies, they show how a simple iterator interface with methods such as `next()`, `hasNext()`, `current()`, `reset()`, and `isReset()` can be used to create simple generators and then combine them into more complex ones. In their work, they test refactoring engines that take programs as inputs, but their techniques are also applicable to testing compilers. We could benefit from their approach when generating more complex patterns than the ones used in our experiments (Section 3.5).

Mutation testing [35, 36, 37] has been long known. It is generally used to evaluate a test suite. The main idea is to create mutants of test cases that behave slightly differently than the original. For a good test suite, mutants are expected to trigger different results than the original program. The ideas are translatable to our work. Our work intends to generate mutants with equivalent behavior and our goal is to evaluate compilers.

Metamorphic testing [38, 39] is a similar approach based on mutations of a test. In

metamorphic testing, one can use a **metamorphic relation** between inputs and outputs of a pair of programs to check for correctness. This can be generalized to a broader context. In the context of performance, for example, the metamorphic relation is that two equivalent programs are expected to have the same execution time.

Differential testing [40] is a technique that compares the output of different implementations of the same problem to determine the correctness of the result. In the context of correctness testing, it is a useful approach when comparing results from different compilers. In the case of performance bugs, we found that compilers vary in the code they generate a lot, and so do the execution times of the programs they compile. In our work, we use differential testing to help compiler writers know how far ahead or behind their compilers are in performance in relation to their peer implementations.

Several works build on top of the idea of generating equivalent mutations to validate compilers through different mutation strategies. Orion [41] generates equivalent programs by deleting statements from dead regions. Athena [42] both deletes and inserts code in dead regions. Hermes [43] inserts code into live regions while ensuring that variable values are preserved afterwards. Our work generates equivalent mutations through semantic-preserving transformations typically performed by compilers.

Studies based on equivalent mutations extend to broader domains such as testing graphics shader compilers. GLFuzz [44] generates equivalent mutations through similar techniques as those used by previous studies, including dead and live code injection, identical arithmetic and boolean expressions, control flow wrapping, and the composition of these transformations. In addition, their work also focuses on test case reduction, through techniques similar to delta-debugging [31], so that the problem is easier to debug by compiler developers.

2.4 HANDLING RANDOMNESS

The main strength of random testing the ability it gives us to statistically quantify the test results [11]. However, to use statistical methods to quantify results, one must ensure that the data and sampling methods meet the requirements of the statistical method being used. For example, when calculating the confidence intervals for means using the T distribution, a method typically taught in undergraduate level statistics classes, one must make sure that data is sampled from a normal distribution [45]. A researcher also needs to precisely describe the context, such as the random seed, in which the experiment is carried out. Finally, a research should repeat experiments to ensure valid results.

A recent study by Klees et al. [13] suggests that the methodologies used to evaluate fuzzers, even ones published at top-conferences, are problematic. First, many of them do

not repeat the experiments to ensure valid results. Second, many of them do not provide enough details on the configuration parameters, such as the random seed to reproduce the experiment. Finally, many of them are not compared against a baseline fuzzer.

Our work repeats experiments and detail all configuration parameters, including, but not limited to, the random seeds and the description of the population of programs being generated. We place high emphasis on the reproducibility of our results, using methodologies described in Section 4.3. However, our work does not have any comparison against a baseline fuzzer. To the best of our knowledge, we are not aware of a fuzzer that focuses on performance, thus we have not evaluated our system against one.

2.5 PROGRAM GENERATION

Our work is based on generating equivalent programs through transformations applied by compilers. The main approaches to program generation are the grammar-based approach and the mutation-based approach.

In the grammar-based approach, programs are generated from a model of the C programming language or its subset. For example, Quest [46] generates C programs that test the correctness of calling conventions. Randprog [47] generates C programs that manipulate volatile variables. Orange [48] focuses on testing arithmetic expressions in C. CSmith [2] is more general-purpose and generates a wide variety of programs while avoiding undefined behavior in C using a combination of static analysis and runtime checks. YARPGen [49] is similar to CSmith, but is able to avoid undefined behavior only through static analysis. Additionally, YARPGen allows the user to specify generation policies, which skew the probability distributions of how programs are generated. Our work restricts generated programs to a small subset of C, while focusing on programs with nested loops and array operations. We avoid out-of-bound errors by checking array sizes and loop bounds with Z3, as discussed in Section 3.7.

In the mutation-based approach, programs are generated by mutating existing programs. Orion [41] deletes code from the dead code regions. Athena [42] not only deletes, but also inserts code into the dead code regions. Finally, Hermes [43] inserts the code into live code regions, but undos the effect of the inserted code later. These studies mutate C programs by parsing them first, using tools such as LibTooling [50]. Our approach generates mutation through semantic-preserving compiler transformations. Since we also generate code, we modify the intermediate representation directly before generating C code.

CHAPTER 3: PERFORMANCE TEST GENERATION

In this chapter, we describe our test generation approach and how we implement it as a test generation system. We discuss the design goals, design choices, and the subcomponents of the system.

3.1 APPROACH

In Section 1.3, we described how one may automatically determine the performance headroom of a program. Namely, when two programs are equivalent, they are expected to have the same runtimes. If one program is slower than the other, we know that it has performance headroom. We can use the fastest runtime among the equivalent programs as the reference runtime when calculating performance headroom.

A program is equivalent to another program when their source code are the same or when one can be transformed into the other through semantic-preserving compiler transformations. A good compiler should be able to undo transformations that cause the program to slow down.

With this notion of equivalence, we can then design a system that tests the performance of compilers using different metrics. For example, equivalent programs are not only expected to have the same runtime, but also the same vector speedup. Metrics can also be developed to compare the effectiveness of different compilers on equivalent programs. Across different compilers, it may be unrealistic to expect them to compile equivalent programs so that they have the same runtime, since such complex systems would have wildly different implementations. However, we believe that metrics for comparing compilers are still useful to give compiler developers an idea of how far their implementation is behind or ahead other compilers.

Since our approach focuses on equivalent programs, our first requirement is to provide ways for the system to generate these programs. The system makes use of compiler transformations to generate programs equivalent to an initial program.

For a given set of transformations, a given program may have a limited number of equivalent programs. For example, a singly nested loop with four iterations can be unrolled at most four times. To test compilers, we need a large number of programs. Besides a large number of programs, we also need a diverse set of programs. The system provides ways to generate programs, and to apply transformations to generate equivalent versions.

Given programs with the same source code structure, compilers may behave differently for

different input sizes. For example, compilers may handle matrix multiplication differently for large and small matrices. The system provides ways to generate code with different data sizes.

Generating tests is only part of the process of evaluating compilers. We also include an iterative script to go over multiple programs (called instances below), transform each one into multiple equivalent programs (called mutations below), build (i.e. compile and link), and execute the C code and finally report the results. The reporting system uses the metrics and statistical methods to report the results.

In summary, we believe performance evaluation of compilers requires an infrastructure that provides users with the functionality to generate equivalent programs for problems of different sizes and is able to provide users with valid, insightful, and reproducible reports. The remaining sections discuss how we designed and implemented the system to meet these goals.

3.2 DESIGN GOALS

Our main design goals include providing users with useful results, making the system easily deployable on different platforms, making the system easy to extend, and making the results reliable.

First, we base our design choices on our intended users. In particular, our target users are compiler developers that would like to know the strengths and weaknesses of their compilers both at a high level and for specific cases. When their compiler performs badly, we believe that concrete examples will help them debug the problem. Our system reports both summary results and identifies outliers.

Next, we try to make many parts of our system platform-independent. We have broken down the system into subcomponents that can be extended easily. For example, we define an internal representation, so that it is possible to target languages other than C by writing a code generation system for each target language.

Next, to increase modularity and extensibility, the system's test generation, code generation, build, and reporting components are decoupled and written purely in Python. An interpreted language like Python is easier to port across platforms than a compiled language. Additionally, using Python and including only packages are easily installed through its standard package manager, Pip Installs Packages (PIP), makes the test generation system easy to deploy.

Finally, since our approach is based on randomness, an inherent challenge that must be faced is ensuring the reproducibility of results in the presence of randomness. Our reporting

system provides basic functionality that allows users to check some requirements for statistical tests. We discuss how we tame randomness and generate reproducible results in some detail in chapter 4.

3.3 SYSTEM OVERVIEW

The overall system consists of several parts as depicted in Figure 3.1. Users interact with the system through an application programming interface (API), configuration files, and command line options. Different parts of the system provide different ways to interact. For example, the test generation subsystem needs high expressiveness and we provide an API for users. The list of compilers and its flags are contained in a Python-based configuration file that can be modified by the user using text editors, if necessary. The build system has a fixed set of options and provides a command line interface.

The pattern generation system is responsible for generating different program structures. The parameters of this system constitutes the pattern profile, which determines the structure of programs being generated. Program structures generated through this process are called **patterns**. Patterns can express a small subset of C functionalities, namely operating on scalars and arrays, arithmetic operations, bitwise operations, logical operations, assignments, and for loops. In particular, the current version of the system does not support functions, pointers, conditional statements, or while loops. Patterns may include constant variables. These are meant to be replaced with literal values in the next stage of the test generation process.

Next, the instance generation system is responsible for generating different problem sizes for a program structure. It does so by replacing constant variables with random literal values specified by the user. These constant variables are typically loop bounds that determine the problem size. Additionally, it determines the array sizes to allocate to handle the computations. Patterns whose constant variables are all replaced and array sizes determined are called **instances**.

Next, the mutation generation system is responsible for generating equivalent programs (called **mutations**) for any given instance. A set of equivalent mutations for a given instance is called a **mutation group**.

Once tests (in the form of mutation groups) are generated, the next step is to generate C code for the mutations. The code generation system generates C code. The code structure later is detailed in section 4.1.4.

The build subsystem is responsible for compiling and linking the different mutations for different compilers and different modes. The user provides a configuration file that contains

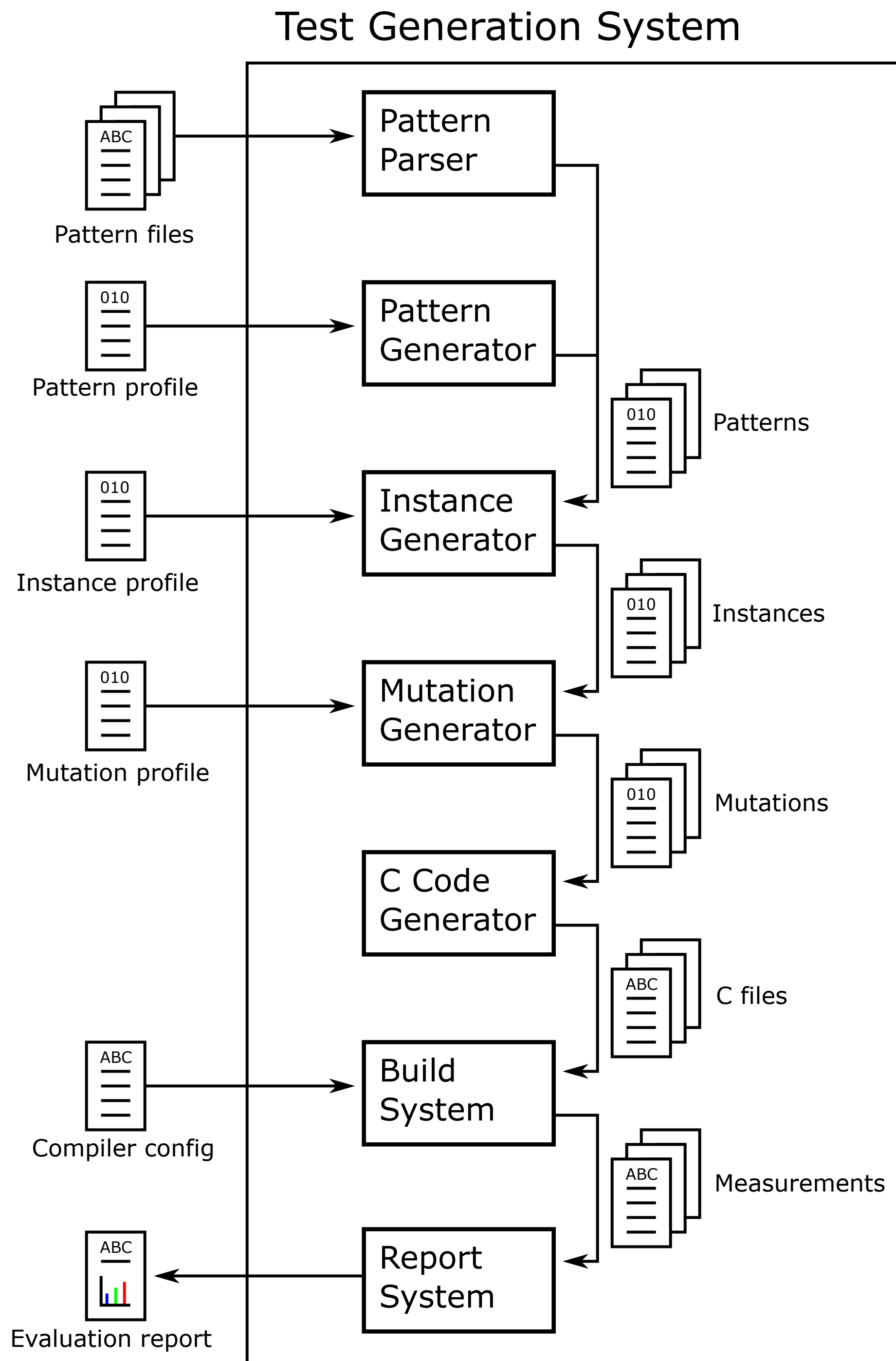


Figure 3.1: The test generation system and its subcomponents

the list of compilers to be used for the experiments, along with their compilation flags for each compilation mode. Compilation modes are discussed more in Chapter 4. The build system also runs the programs, checks for correctness, and saves measurement results.

Finally, the reporting subsystem generates compiler evaluation reports. This is specific to how we evaluate our work and is described in detail in Chapter 4.

3.4 RANDOM TEST GENERATION

Whenever we are dealing with random samples, it is important to define the population from which we are sampling. We use **profiles** to describe populations. Profiles can be formal or informal, as long as it is clear to the reader what population they are describing.

The main sources of randomness in our work include the pattern, instance, and mutation generators that work together to generate random tests. We have three different profiles to describe each of them. A pattern profile describes the population of patterns. An instance profile describes the population of instances. A mutation profile describes the population of mutations. We provide more details for each of the profiles when we describe each subsystem.

We use the uniform distribution whenever randomness is needed. This means for a given set, the probability of any element being chosen from that set is the same. We use the default random seed in Python by not explicitly setting the seed at all.

3.5 PATTERN GENERATION

The space of all possible patterns, even if we limit the size of the pattern, is immense. While our intermediate representation can express more complex patterns, we chose to start our studies with simpler patterns that have simple structures. In particular, we enforce the following pattern structure.

- A program is always a sequence of loops.
- All loops have the same depth, are perfectly nested, and have the same number of statements inside.
- All statements inside the perfectly-nested loops are assignments.
- All right hand side of the assignments have the same number of operators.
- All indices are affine functions of a single loop variable with the form `coeff * loop_var + const` or `coeff * loop_var - const`.

The pattern generator takes as input a **pattern profile** and randomly generates patterns according to the profile. The pattern profile allows users to specify several aspects of the pattern structure through sets of possible values or numbers. When the pattern generator reaches a point where randomness is needed, it picks a choice from these sets of possible values as an answer. Properties of the pattern profile are shown in Table 3.1.

Parameter name	Meaning
Array	The set of possible array names that may appear in the pattern, along with the number of dimensions.
Coeff	The set of possible constant variable names that may appear as the coefficient of the affine array indices. These constant variables are intended to be replaced by positive integers.
ZeroCoeff	The set of possible constant variable names that may appear as the coefficient of the affine array indices. These constant variables are intended to be replaced by non-negative integers.
Const	The set of possible constant variable names that may appear as the constant term of the affine array indices. These constant variables are intended to be replaced by non-negative integers.
Data	The set of possible constant variable names that may appear as operands of the right hand side computations.
LoopVar	The set of possible loop variable names.
# Loops	Number of loops in the program. Nested loops count as one loop.
# Stmts	Number of statements in each loop in the program.
# Ops	Number of operations appearing on the right hand side of each assignment.
LoopDepth	The depth of each loop in the program. All loops have the same depth.
Ops	The set of possible operations that may appear on the right hand side of each assignments.

Table 3.1: Parameters of a pattern profile

The pattern generation process starts by determining how many loops need to be generated. This is read from the # Loops property. For each loop, the loop variables to be used are chosen randomly from **LoopVar**. The number of statements inside the loop is determined by # Stmts.

For each statement, on the left hand side the system picks an array whose element is to be assigned to. An array may have zero dimensions, in which case, it is considered a scalar

variable. On the right hand side, an expression is generated with # Ops number of operators. Operators are picked at random from **Ops**. The operands are either array elements or some constant variable picked from Data.

If an array access needs to be generated, the system randomly picks an array and generates affine indices for each of the dimensions of the array. The affine indices either have the form `a*i + b` or `nz*i - b`. Where $i \in$ LoopVar, $a \in$ Coeff $\uplus$ ZeroCoeff, and $nz \in$ Coeff.

In our studies, we restrict our indices to non-negative values to simplify the implementation, even though negative indices are not illegal in C. We also assume that negative indices are not typically used in programs.

When we encounter negative indices, we discard that test and try to generate a new one. During our experiments, we noticed a lot of time wasted because of negative indices. To improve test generation speed, we split coefficients into **Coeff** and **ZeroCoeff**. When a variable name from ZeroCoeff is chosen to be the index coefficient, the affine index will never be in the form `nz*i - b`. Without separating **Coeff** and **ZeroCoeff**, negative indices such as $0 * i - b$ would happen more frequently, and tests would need to be discarded more often.

Note that it is up to the instance profile (described in section 3.7) to ensure that range of **Coeff** does not include zero. It is also up to the instance profile to ensure that **Const** does not range over negative values.

Table 3.2 shows an example of a pattern profile, and Figure 3.2 shows an example pattern that can be generated from it.

Property	Value
Array	{A(1), B(1), C(2), D(2), E(3), F(3)}
Coeff	$\{a_1, a_2\}$
Const	$\{b_1, b_2\}$
Data	$\{f_1, f_2\}$
LoopVar	$\{i_1, i_2\}$
# Loops	1
# Stmts	2
# Ops	2
LoopDepth	2
Ops	$\{+, *, -\}$

Table 3.2: Pattern profile

The **declare** keyword declares variables and their dimensions. The **for** keyword is used to define perfectly nested for loops, along with the name and ordering of loop variables. Array accesses, assignments, and operations share the same syntax with C. The rest of the syntax

```
declare A[];
declare C[][];
declare D[][];
declare E[][][];

for [i1, i2] {
  A[a1 * i2 - b1] =
      E[z1 * i1 + b1][a1 * i2 - b1][z2 * i1 + b1] *
      ( C[z2 * i2 + b2][z2 * i1 + b1] +
        A[a2 * i2 + b1]
      );
  E[z2 * i2 + b1][a1 * i1 + b1][a1 * i2 + b1] =
      D[a2 * i2 - b1][z2 * i1 + b1] -
      ( f1 +
        A[z1 * i1 + b1]
      );
}
```

Figure 3.2: Generated pattern using the pattern profile from Table 3.2

```
declare A[409];
declare C[6][15];
declare D[351][15];
declare E[409][409][379];

for [(i1, >=132, <=394), (i2, >=143, <=364)] {
  A[1 * i2 - 14] =
      E[1 * i1 + 14][1 * i2 - 14][0 * i1 + 14] *
      ( C[0 * i2 + 5][0 * i1 + 14] +
        A[1 * i2 + 14]
      );
  E[0 * i2 + 14][1 * i1 + 14][1 * i2 + 14] =
      D[1 * i2 - 14][0 * i1 + 14] -
      ( 0.9955111516629354 +
        A[1 * i1 + 14]
      );
}
```

Figure 3.3: Generated instance using the pattern from Figure 3.2

is a small subset of C. The full syntax is shown later in Figure 3.4.

Note that not array appear in the pattern, and that different arrays have different dimensions as described by the pattern profile. For all affine indices, if z1 or z2 is the coefficient, the constant term will only be added, while if a1 or a2 is the coefficient, the constant term can either be added or subtracted. Data constants, such as f1 also appear in the patterns and only on the right hand side of the assignments.

3.6 FIXED PATTERNS

While patterns can be generated, the system is designed to take fixed patterns from the user as well. This is useful when the user needs more control of the pattern population, such as studies about real world patterns that may be hard to capture through pattern profiles. For this, the system provides a concrete syntax so users can directly write patterns, with the same syntax as the example code in Figure 3.2, into files and use the system's API to parse the patterns. The pattern language core syntax is shown in Figure 3.4.

A pattern file declares variables and their dimensions if they are arrays. Users can optionally specify the sizes in each dimension of the arrays. Statements follow declarations. Statements are either for loops or assignments. In for loops, loop bounds and steps are optional. For convenience, users may specify loop nests through a single for-loop. Expressions include scalar access, array access, literals. Expressions also include a subset of C operations, namely conditional, logical, bitwise, and arithmetic expressions with the same operation precedences defined by C. The language does not include pointers, while loops, or conditional statements.

3.7 INSTANCE GENERATION

Patterns specify the program structure in an abstract way. They may not specify loop bounds, coefficients, constant terms, and literals that may be used in computations. The array sizes are also undetermined. These unknown quantities can be made concrete so that patterns can be **instantiated** for different problem sizes. The instance generator is responsible for randomizing different parameters that determine the problem size.

The randomization of coefficients, constant terms, and literals are straightforward. The system uniformly picks a number of the possible range of values and replaces the names in the abstract syntax tree with the actual values. The instance generator has an added responsibility of determining the array sizes that are needed to carry out the computation.

```
// Pattern program
program ::= decl+ stmt+

// Declaration
decl    ::= "declare" var dim* ";"
var     ::= IDENTIFIER
dim     ::= "[" INT? "]"

// Statements
stmt    ::= loop | assignment

// For loop
loop    ::= "for" "[" header "]" "{" body "}"
header  ::= var | "(" shape ")"
shape   ::= var begin? end? step?
begin   ::= "," ">=" expr
end     ::= "," "<=" expr
step    ::= "," "+=" expr
body    ::= stmt+

// Assignment
assign  ::= access "=" expr ";"
access  ::= var index*
index   ::= "[" expr "]"
expr    ::= access
          | ... // literals and operators omitted for brevity
```

Figure 3.4: Syntax of program patterns

Our implementation uses the Z3 SMT solver [51] as the backend for finding array sizes. Users have the choice to specify the array size or let the system determine it. Given affine functions of loop variables along with the loop bounds, it is possible to encode array sizes as the maximum value of an affine function, which Z3 can solve. For example, if there is an array access `A[2*i-5]` in the program, and we know that loop variable `i` ranges over [0, 10], then the array size must be able to hold at least $2 * 10 - 5 = 15$ elements.

The system also uses the solver to determine whether invalid indices can occur or not. Indices are invalid when they are negative or out of bounds. For the same example, we add the constraint $2 * i - 5 < 0$ and check for satisfiability. In this case, it is possible to have a negative index, such as when i is 0, and therefore we discard this instance. Suppose the user defined the array size to be 10, then this is an invalid instance, since the maximum index

can be up to 15, which is out of bounds.

Finally, the system uses the solver to check the presence of loops with no iterations. In particular, we check that for every loop variable i that $i_{low} \leq i \leq i_{high}$ holds while also satisfying the previous constraints, such as non-negative indices. If the loop bound constraints are unsatisfiable, it means that it is not possible for a loop variable to have a value between i_{low} and i_{high}, which in turn means that valid iterations do not exist.

The main reason for using Z3 is mostly a matter of convenience. Its use made the implementation of the system quicker. Another advantage that Z3 (and several SMT solvers such as MCSat [52]) has over linear integer programming libraries, such as the Integer Set Library (ISL) [53], is that Z3 supports non-linear constraints. Although the theory of non-linear integer arithmetic is undecidable [54], Z3 is still able solve specific instances. However, in our studies, we did not generate any non-linear integer constraints as all our indices are affine functions on a loop variable, and we realize this is not a strong argument for using Z3.

Like patterns, instances also have profiles. They randomize values to substitute for the constant variables that appear in the patterns. Table 3.3 shows the parameters of an instance profile.

During substitution, the system blindly substitutes values for constant variables. After substitution, the system checks whether an instance is valid or not. An instance is invalid when an array can have invalid indices or a loop can have no iterations. If an instance is invalid, the system discards the instance and tries substituting different random values. In our studies, when the system retries up to a certain number of times, with the default being 10,000, the system discards the whole pattern and regenerates a new pattern.

Table 3.4 shows an example of a pattern profile, and Figure 3.3 shows an example instance that can be generated when using it with the pattern from Figure 3.2.

Note that the arrays have sizes associated with them, all coefficients, constant terms, and data constants have been replaced with literal values, and the loop bounds are determined. The array sizes are automatically computed so that they are large enough to hold all the indices used in the computation.

3.8 MUTATION GENERATION

Measuring compiler stability relies on having equivalent programs to run and measure. A mutation group includes equivalent mutations that are generated from the same instance. We can generate these mutations through automatic source-level transformations that compilers typically perform.

We use the term **source-level transformation** loosely. In particular, although our final

Parameter name	Meaning
Coeff	The range of values [x, y] that can replace constant variables that belong to the set Coeff in the pattern profile. We expect x and y to be positive integers, where $x < y$.
ZeroCoeff	The range of values [x, y] that can replace constant variables that belong to the set Coeff in the pattern profile. We expect x and y to be non-negative integers, where $x < y$.
Const	The range of values [x, y] that can replace constant variables that belong to the set Const in the pattern profile. We expect x and y to be non-negative integers, where $x < y$.
Data	The range of values [x, y] that can replace constant variables that belong to the set Data in the pattern profile. We expects x and y to be either integers or single-precision floating points, where $x < y$
$\text{LoopVar}_{\geq}$	The range of values [x, y] for the lower bound of a loop. In the loop header `for (int i = c; i <= N; i += ...)`, this value replaces `c`. We expect x and y to be non-negative integers.
$\text{LoopVar}_{\geq}$	The range of values [x, y] for the upper bound of a loop. In the loop header `for (int i = c; i <= N; i += ...)`, this value replaces `N`. We expect x and y to be positive integers. The lower bound of ranges for $\text{LoopVar}_{\geq}$ is expected to be larger than the upper bound of ranges for $\text{LoopVar}_{\geq}$
$\text{LoopVar}_{+=}$	The range of values [x, y] for step size for the loops. We expect x and y to be positive integers. When unspecified, defaults to [1, 1].
ArraySize	(Optional) The names and sizes of the arrays. We expect the sizes to be a list of positive integers. When unspecified, the system determines the array sizes for the user.

Table 3.3: Parameters of an instance profile

Set	Range of values
Coeff	[1, 2]
ZeroCoeff	[0, 2]
Const	[0, 16]
Data	[0.5, 2.0]
$\text{LoopVar}_{\geq}$	[100, 200]
$\text{LoopVar}_{\leq}$	[300, 400]

Table 3.4: Instance profile

code is in C, we do not perform source-level transformation at the C-level. Rather, we perform transformations on an internal representation, which is the abstract syntax tree (AST) of the pattern language. We then generate C code from this internal representation, as described in Section 3.9.

Each source-level transformation mutates the AST in different ways. For example, the loop unroll and loop unroll-and-jam transformations unroll the statements in the body of the original loop, increases the step size in the loop header, and adds new statements to carry out computations in the remainder of the unrolled loops. On the other hand loop interchange simply swaps the ordering of the loops by reordering the loop variables in the loop header.

Source-level transformations may depend on knowledge of the data dependences in the program. For example, the validity of loop interchange depends on making sure data dependence does not change, such as reversing a read-after-write to a certain array element. Our work includes a dependence analysis system, based on determining the satisfiability of equations and inequations. Again, we use Z3 to implement dependence analysis through checking for satisfiability of accesses to the same array element. For example, we can check whether an array accesses A[i] and A[i-1] can ever refer to the same element or not by representing the first access as i_1 and the second as $i_2 - 1$, then check if $i_1 == i_2 - 1$ is satisfiable, given other constraints such as loop bounds and the ordering of the iterations. The current implementation assumes that indices are always a function of the loop variables. This means one of its limitation is the inability to detect and eliminate induction variables.

During the implementation of a dependence analysis system with an SMT solver (Z3) and an interpreted language (Python), we have learned that with modern hardware, it is not time-prohibitive for our studies which include only small codelets. For each study we spend less than 60 minutes to generate 1600 mutations, which entails performing dependence analysis hundreds of times. Most of the time was spent on discarding invalid instances and retrying. In the algorithm, we do not distinguish different subcases, such as handling zero or single or multiple subscripts differently as described by Allen and Kennedy [55]. We were able to afford to make our code succinct and lightweight by always treating every case as multiple subscripts, which is the most general case. We push the complexity for the algorithm to Z3 to handle. Our dependence analysis algorithm includes less than 500 lines of Python code, including comments.

Different source-level transformation have different random choices in different transformation spaces. For example, for loop unroll, the size of the transformation space depends on the loop bound, and the unroll factor can be specified using a single integer. On the other hand, the transformation space for loop interchange depends on the depth of the loop nest,

and while it is possible to linearize the ordering of loops into a single integer, expressing the mutation profile as the ordering of loops is more intuitive for the reader. It is difficult to reconcile the different transformation spaces for different transformations into a single set of properties that describe the mutation profile. Here, we chose to describe the mutation profile informally. In the code, this appears as calls to different methods for different types of classes.

When one studies a sequence of transformations, the transformation space becomes even more complicated. Suppose we have a doubly-nested loop, and we would like to tile it first, then permute it next. If we tile only one of the loops, the doubly-nested loop will become a triply-nested loop and there will be 6 possible permutations. However, if we tile both loops, we will have a loop nest of depth 4, and we will have 24 possible permutations. The space for each decision is not uniform. Not only does this make it more difficult to describe a unifying mutation profile for all sequences of transformations, but it raises another question about the randomization process. Suppose we chose to tile only one loop, there are 2 ways to tile it, and once we tile one loop there are 6 ways to permute it, totaling 12 mutations. On the other hand, if we choose to tile two loops, we will have 24 loop permutations afterwards, totaling 24 mutations. Disregarding the tile sizes, which would complicate the situation further, we have a transformation space including $12 + 24 = 36$ permutations. We would then need to decide whether we would like to choose from this space uniformly or make uniform choices once during tiling and once again during permutation. These decisions result in different probabilities of each mutation. At the end, it would depend on what mutation population the user wants to describe.

3.9 C CODE GENERATION

Since the functionality of the pattern language is a small subset of the C language, it was simple to generate C code for the patterns. Given mutations from the mutation generator, the C code generation system generates C code and writes it to the file system. It also generates a list of files for the build system to use as its input. Details on how we divided the tests into translation units in C are included in section 4.1.4.

3.10 BUILDING AND RUNNING EXECUTABLES

The goal for the test generation system is mainly to provide an infrastructure for compiler developers and researchers to evaluate compilers. Our evaluation, detailed further in Chapter 4, is based on compiling the same source code with different compilation flags and

compilers. The build system reads the list of files from the C code generation system and manages building different executables for each of the compiler configurations.

Once executables are built, the build system first runs them to check for correctness. Any pattern with incorrect mutations, no matter which compiler and flags it was built with, is excluded from the report by maintaining a pattern blacklist.

Next, the build system attempts to approximate how many nanoseconds it takes per one iteration of the kernel. It does this first running the executable for one iteration. This single iteration may have delays from the cache miss. Next, it uses this number to approximate the number of iteration it needs to run for one millisecond. It bounds this number to be between 1 and 100. It then runs the executable with the bounded number to find the average nanosecond per iteration.

Next, the build system runs each of the executables with the number of iterations so that the test will either take more than 100 milliseconds or run between 1 and 100 iterations. The runtimes of each iteration is saved in a file. More details on how we measure the runtime in included in section 4.1.7.

For the implementation of the build system, we used the `doit` workflow management library, which is included in the standard Python package manager. It integrated well with the whole system which is written in Python.

3.11 REPORTING RESULTS

Finally, once we have all the executable files, the reporting system reads the runtimes that are saved from the build system and produces an evaluation report. Reporting valid and reproducible results in a random system is one of our main goals and requires a lengthy discussion. We defer the discussion on reporting results to Chapter 4.

CHAPTER 4: EVALUATION

In this chapter, we evaluate the usefulness of our test generation approach. We conduct multiple different studies using the test generator described in chapter 3. As can be seen throughout this chapter, the system greatly facilitates the implementation of experiments for finding performance bugs of modern compilers.

To study the effectiveness of our test generator in finding performance bugs, we considered four modern compilers on how well they handle different automatic source-level transformations. We generate multiple tests, have each compiler translate the tests, measure the run times, and use our metrics to describe how well each of the compilers do their job in absolute terms and relative to each other.

The source-level transformations studied in our work include loop unroll, loop interchange, and loop unroll-and-jam. More transformations can be added to the system in a straightforward and modular manner. However, while studying the effects of other transformations is an interesting topic, the focus of this thesis is the methodology of finding performance bugs and conducting reproducible experiments in a random system.

This chapter is organized as follows. Section 4.1 discusses how our experiments are set up in detail. This step is crucial due to the nature of our work. We are taming randomness, and therefore, we need to describe exactly what is controlled and what is random. Section 4.2 discusses the metrics we use in our studies. There are several metrics and each serve different purposes. It is important that the reader understand what each is trying to achieve and interpret the results correctly. Section 4.3 discusses how we use statistical methods to quantify randomness and achieve reproducible results. Finally, sections 4.4, 4.6, 4.7, 4.8, and 4.9 discusses our experiments and how we used the test generator to evaluate compilers. This demonstrates the usefulness of our system, and hopefully motivates readers who is curious about similar other evaluations to use the system and conduct studies that may suit their needs.

4.1 EXPERIMENT SETUP

4.1.1 Hardware

All experiments are run on dedicated hardware, as opposed to a shared cluster or virtual environment, with the Intel Xeon W-2195 2.30GHz processor. This processor supports the AVX512 instructions. We turned off hyper-threading [56] and turbo boost [57] to avoid

unreliable performance measurements.

During our experiments, we made sure no other CPU-intensive programs were running and that only the user running the experiments was logged into the system. Experiments were run serially, one at a time. We did not control any of the the running system services. It is possible that CPU spikes may occur because of system services. We believe they do not affect performance measurements of the experiments as a whole, since our sample size is large. The few programs affected by the system noise may have higher runtime variation and will likely appear as outliers for the metrics we use (Section 4.2). The system reports outliers in the evaluation results. Users can manually inspect programs with suspicious runtimes and may decide to re-run them.

4.1.2 Compiler Versions

The compilers we include in our experiment are C compilers with vectorizing capabilities. We study the behavior of four vectorizing C compilers. The compilers and the versions are shown in table 4.1

Compiler	Version	Release date
Clang	10.0.0	March, 2020
ICC	19.0.4	April, 2019
GCC	7.5.0	November, 2019
PGI	19.10	November, 2019

Table 4.1: Compiler versions

4.1.3 Compiler Modes and Flags

We evaluate compilers on their stability metrics, which aim to measure runtime, vectorization capabilities, and accuracy of the cost model. These metrics make use of three compilation modes: fast, no-vec, and no-predict.

The main compilation mode used by every metric is called fast mode. Fast mode enables the highest level of optimization provided by the compiler. Typically, this uses the "fast" flag. Some compilers need additional flags to specify the vector width preference.

These aggressive optimization flags may enable other sub-flags related to optimizing floating point operations that do not fully conform to the ANSI standard. For example, ICC's documentation warns that the floating point optimizations may "affect the accuracy or reproducibility of floating-point computations". GCC is more descriptive and lists the optimizations performed, including allowing associativity of floating points, ignoring the sign

Compiler	Fast mode
Clang	`clang -Ofast -march=native -mprefer-vector-width=512`
ICC	`icc -Ofast -xCORE-AVX512 -qopt-zmm-usage=high`
GCC	`gcc -Ofast -march=native`
PGI	`pgcc -fast`

Table 4.2: Fast mode commands for each compiler

of zeros, and using reciprocals instead of divisions. Clang includes the same flags as GCC. PGI does not detail their handling of floating points.

The no-vec compilation mode tells the compiler to perform everything that fast mode does, except that vectorization is turned off.

The no-predict compilation mode tells the compiler to perform everything that fast mode does, except that the cost model is turned off. When the cost model is turned on, compilers will vectorize code when it is deemed profitable by the cost model. When the cost model is turned off, compilers will always vectorize code when possible without consulting the cost model.

The flags for no-vec and no-predict mode are listed in table 4.3. When used, these flags are appended to the existing flags of fast mode.

Compiler	No-vec mode	No-predict mode
Clang	`-fno-vectorize`	`-mllvm -force-vector-width=16`
ICC	`-no-vec`	`-vec-threshold0`
GCC	`-fno-tree-vectorize`	`-fvect-cost-model=unlimited`
PGI	`-Mnovect`	`-Mvect=nosizelimit`

Table 4.3: Additional flags for no-vec and no-predict mode

Each compiler handles the cost model differently. The flags for no-predict mode have more variation and requires further explanation. For Clang, to disable the cost model, we force the vector width. Otherwise, the compiler may choose other vector widths, including not vectorizing the code at all. For ICC, the vectorization threshold tells the compiler to vectorize when the probability of speedup determined by the cost model is above the threshold. Threshold 0 means even if the cost model determines that the code will not benefit from vectorization, vectorize the code regardless. GCC defines the unlimited cost model to always assume that vectorizing is profitable. Finally, PGI determines whether to vectorize loops based on the number of statements in the loop. Setting this to nosizelimit, means that the compiler will always vectorize the loop regardless of the number of statements

in the loop.

4.1.4 Program Structure

In this section, we describe the structure of the generated programs. In order to find performance bugs in a large and complex system like compilers, we expect to be generating a large number of tests in hope to detect cases where compilers perform poorly. We believe it is better for each test (i.e., mutation) to be a separate executable file rather than having a single file contain all generated tests. It makes it easier for users to study specific tests further.

Each mutation contains three units of compilation, each contained in separate files: the main function, instance-specific boilerplate code, and the kernel. We made sure link time optimization was not enabled for any of the compilers to make sure it does not unintentionally interfere with the compilation flags for each mode.

The main function source file handles command line arguments so users can specify whether the test should print the result of the computation or the performance values. While our main focus is measuring performance, it is our due diligence to check for correctness as well. Each generated test also includes functionality to print the result of the computation so that the mutation may be checked for correctness. We detail how correctness is checked and how performance is measured in sections 4.1.6 and 4.1.7, respectively. The same main source file is used for all tests since its functionality is not dependent on the array size or the computation being carried out.

The instance-specific function handles initialization of the arrays, printing the output so it can be used to check results, and printing the runtime so it can be used to measurement performance. The instance-specific source file is shared across mutations, since any mutation of the same instance uses the same arrays and these arrays have the same sizes. Arrays are allocated on the heap to avoid stack size limitations. Because we use the heap, we may miss opportunities to catch compiler bugs with large stack sizes, but using the heap allows the system to include performance tests with very large arrays. For data initialization, the system always initializes the random seed to 0 and initialize elements in the array with single-precision floating point values between 0 and 1. Arrays are passed to the function containing the kernel by reference. Scalars are passed by value.

The kernel source file contains the function that carries out the computation. The computation is surrounded by timers in order to measure the runtime. The kernel function returns the time it takes to finish the computation in nanoseconds. All arrays passed to the kernel are passed with the keyword `restrict`.

Separating each kernel function into a single file has several benefits. First, it simplifies debugging. Since the assembly code for the function is well-contained in one file. Second, with link time optimization turned off, the kernel is always compiled and called. Even if the results are not used later on, the compiler will not deadcode eliminate the computation. As a result, we will still be able to measure the kernel's performance even if the output is not printed. Another alternative to achieving the same results is with dummy functions that forces the compiler to assume that the output of a kernel is used.

The downside to this approach is that all kernels are contained in a separate function. To study how a kernel's performance may interact with its surrounding code, we would have to use a different strategy. We have not considered this problem in detail, but since our approach is random testing, we would have to control the random noise, including performance variation resulting from surrounding code, as much as possible. This would make the extension that includes the context a challenging problem.

4.1.5 Linking and Running the Code

While independent tests can be run in parallel for correctness testing, running multiple tests at the same time may affect runtime measurements for performance testing. However, it is still possible to parallelize compilation and correctness checking. We build all tests and check their correctness in parallel, but measure the runtimes sequentially.

When generating tests for compilers, one may expect to generate a large number of tests since compilers are large and complex systems. In our study, we attempt generate the right amount of tests to balance between revealing interesting behaviors of compilers, returning reproducible statistical results, and not taking too long.

Each experiment discussed in this thesis show interesting performance behaviors, are reproducible, and took no longer than 60 hours to build, check correctness, and measure runtimes.

4.1.6 Checking for Correctness

The test generator focuses on generating programs with intensive single-precision floating point calculations, since high-performance computing typically focuses on them. This poses several challenges when checking for correctness.

First, round off errors can happen since we allow associativity of floating point numbers when transforming reductions. This may lead to different results because of round-off errors. We must account for them and cannot simply compare results using equality. Handling

round-off errors in floating-points have been well-studied [10, 58]. Automatic solutions, such as automatic round-off error estimation through static analysis [59] or abstract interpretation [60], have also been proposed. We did not adopt these techniques in our work due to limited time.

Second, tests are generated at random. It is possible, and perhaps more likely, to generate programs that result in values including inf, -inf, and nan. We have to account for these values.

Finally, we find it desirable to store the results of each test onto disk. This helps us detect and avoid re-computation of results for experiments with large number of tests that may be interrupted any time. Since the arrays used in the tests may be large. Storing the values of the output arrays for each test onto disk may be space-prohibitive.

We mitigate these problems through several design choices. We combine approaches from the TSVC benchmark [8] and differential testing [40]. We summarize the output of a test by adding all elements in the array together. We call this the checksum of the program. For any given mutation, we can always find another mutation that returns the same checksum. Specifically, other mutations in the same mutation group of a given instance, the same mutation compiled in by the same compiler in other modes, and the same mutation compiled by other compilers should all return the same result. As described in 4.1.4, the random seed to initialize data is always 0. The checksum for different programs should return the same result.

To work around analyzing precise round-off errors through numerical methods, we simply consider a 1% deviation around the median checksum as correct.

To mitigate the problem with inf, -inf, and nan values, our studies only initialize array elements with numbers between 0 and 1. Additionally, we try to avoid large floating point literals and iteration counts when we define the instance profiles. This reduces the chance that results of a computation will reach inf or -inf. When computing the checksum, if we encounter non-normal numbers, we treat them as 0.1.

These workarounds are not perfect. For example, swapping elements in an array result in different arrays, but the checksum method will consider them the same. Treating non-normal numbers as 0.1 uniformly means that inf, -int, and nan are considered the same, when they are actually not. Two inf values may not be the same either since they may have arrived from different calculations.

When a test is incorrect, there are several possibilities. It may be a bug in our code generation process, it may be a bug in the source-level transformer, or it could actually be a bug in the compiler we are evaluating.

For each new feature in the system or new source-to-source restructuring modules, checking

for correctness has helped us find bugs in the system, which have all been fixed before running the experiments reported. We even found a confirmed bug in the Z3 SMT solver that we use as the backend for our dependence analysis, and that has been fixed as well. We have high confidence in the correctness of the system used in all the studies in this work.

Incorrect programs are reported to the user and omitted from reporting performance results.

4.1.7 Measuring Runtime

One source of random noise when measuring runtime is from the target system. For example, a service running in the background may suddenly use more CPU than normal and happen to use the same CPU as our tests. Additionally, the reliability and precision of the system clock affects how consistent we can measure runtime.

We use `clock_gettime` with `CLOCK_MONOTONIC` to measure the runtime of the kernels. A call to `clock_gettime` on the system we used takes about 20 nanoseconds when we average a million calls in a row. For code that runs very fast, we found that `clock_gettime` can consistently time code that runs as fast as 50 nanoseconds per iteration.

Instructions such as RDTSC may be considered for extremely high precision hardware clocks. Since `clock_gettime` worked reliably on our system, we did not pursue RDTSC. Another point to note is that if we would like our kernels to be portable to systems that are not POSIX-compliant, we would have to wrap the timing function in another function that is system dependent.

When measuring runtime, we repeatedly run the kernel and take the arithmetic average. We either run the kernel for 100 iterations or run until it reaches 100 milliseconds, whichever uses less time. This averages out the noise and the affect of filling up both the data and instruction cache during the first iterations. For program running longer than that, we assume that the affect of the cache during the first iteration is negligible compared to the overall runtime.

In our studies, we try to generate tests so they take no less than 1 millisecond to reduce the effect of system noise further.

4.2 METRICS

In this section, we formally define the metrics quality of compiling for performance in our study. Since our study is based on the notion that stability reflects performance, most of our

metrics are based on stability. On the other hand, stability is only one facet of performance. Therefore, our metrics also include comparative speedups across compilers.

The organization of this section is as follows. First we define general notations that are used to define the metrics formally. Next, we describe each metric, their rationale, and their definition. Finally, we discuss some alternative metrics that we did not use and the reason why we did not use them.

4.2.1 Notations and Helper Functions

While we only have a single measurement which is the runtime of the kernel, in order to define our metrics, we need to specify which compiler, compilation mode, pattern, instance, and mutation the runtime is for. We define the following notations to help define the metrics later on.

All collections defined through curly brackets are multisets, meaning that they allow multiple instances of the same element. This is important since multiple instances of the same value make a difference when calculating the average of a collection.

When defining metrics, we use naming conventions to determine the variable type. The variable name c identifies a compiler. The variable name p identifies a pattern. The variable name i identifies an instance. The variable name m identifies a mutation. The variable name r identifies a runtime. If there are multiple variables of the same type, they will be subscripted with numbers. Names of multisets or functions that return multisets are capitalized and bold, while names of scalars or functions that return scalars are uncapitalized and italicized. For example, $\mathbf{F}(x)$ is a function that returns a set, and $g(x)$ is a function that returns a scalar.

In a given experiment, $\mathbf{Compilers}$ is the set of compilers under study, $\mathbf{P}$ is the set of all pattern names. The set $\mathbf{I}(p)$ denotes all instances generated from pattern p. The set $\mathbf{M}(i)$ denotes all mutations generated from instance i.

The function $r_{\text{fast}}(c, p, i, m)$ denotes the runtime of pattern p, instance i, mutation m, when compiled by compiler c in the fast compilation mode. Similarly, we define r_{novec} and $r_{\text{nopredict}}$ that accepts the same list of parameters for the no-vec and no-predict compilation modes, respectively.

The functions *min* returns the mininum value of a set, and *max* returns the maximum value.

The arithmetic mean (*mean*) and geometric mean (*geomean*) of a set $X = \{x_1, ..., x_N\}$ are calculated as follows.

$$mean(X) = \frac{1}{N}\sum_{i=1}^{N} x_i \tag{4.1}$$

$$geomean(X) = \sqrt[N]{\prod_{i=1}^{N} x_i} \tag{4.2}$$

4.2.2 Stability Metrics

Stability metrics include the runtime stability, vector speedup stability, and cost model stability. These metrics gauge the variation in runtime across mutations and different compilation modes. They give users a sense of how consistent in performance a compiler is when translating a set of equivalent source codes.

4.2.3 Runtime Stability

A perfect compiler can compile all mutations in the same mutation group to executables with matching runtimes. In addition, that runtime should be the best that the compiler can do. We use this notation to quantify performance headroom. For example, if two mutations from the same mutation group have runtimes of 100 nanoseconds and 200 nanoseconds, we know that there is performance headroom for the mutation that runs slower.

A perfect compiler can detect all equivalent programs generated by source-level transformations. Such a compiler is unlikely to exist, since the space to undo those transformations is large, and exhaustively searching all equivalent programs is impractical. However, in reality, our expectation would be that a compiler would perform relatively well. The runtime stability quantifies how well a compiler does.

All mutations of the same instance are equivalent programs. If we know that a compiler can compile a mutation to run very fast, we should expect that it can compile other mutations to run very fast as well. We can use the best-performing mutation as the reference runtime. The reference runtime is always less than or equal other runtimes in the same mutation group, and we can use it to scale the runtimes so their values lie between 0 and 1. With these ideas, we formally define the scaled runtime and runtime stability as follows.

$$\mathbf{R}_{\text{instance}}(c, p, i) = \{r_{\text{fast}}(c, p, i, m) \mid m \in \mathbf{M}(i)\} \tag{4.3}$$

$$r^{\text{min}}_{\text{instance}}(c, p, i) = min(\mathbf{R}_{\text{instance}}(c, p, i)) \tag{4.4}$$

$$r^{\text{scaled}}(c,p,i,m) = \frac{r^{\text{min}}_{\text{instance}}(c,p,i)}{r_{\text{fast}}(c,p,i,m)} \tag{4.5}$$

$$r^{\text{stability}}_{\text{instance}}(c,p,i) = geomean(\{r^{\text{scaled}}(c,p,i,m) \mid m \in \mathbf{M}(i)\}) \tag{4.6}$$

$$r^{\text{stability}}_{\text{pattern}}(c,p) = geomean(\{r^{\text{stability}}_{\text{instance}}(c,p,i) \mid i \in \mathbf{I}(p)\}) \tag{4.7}$$

$$r^{\text{stability}}(c) = geomean(\{r^{\text{stability}}_{\text{pattern}}(c,p) \mid p \in \mathbf{P}\}) \tag{4.8}$$

For a given compiler c, the meaning of each equation is as follows. Equation 4.3 gathers runtimes for a mutation group. Equation 4.4 gets the minimum value from $\mathbf{R}_{\text{instance}}$ to use as the reference runtime for that mutation group. Equation 4.5 scales the runtime of each mutation so that its value will fall between 0 and 1, with numbers closer to 1 signifying being closer to best runtime in its mutation group. Equation 4.6 computes the runtime stability for an instance by averaging the scaled runtimes of mutations that belongs to the same instance. Equation 4.7 computes the runtime stability for a pattern by averaging the runtime stability of instances that belong to the same pattern. Finally, Equation 4.8 computes the runtime stability of a compiler by averaging the runtime stability of all the patterns.

In summary, the runtime stability indicates on average what is the slowdown of each mutation when compared to the best mutation in the same mutation group.

4.2.4 Vectorization Speedup Stability

Similar to our expectations for runtime, for a perfect compiler, when given different mutations in the same group, the speedup gained from vectorization would be the same. Here, we define the vectorization speedup function v as the no-vec mode runtime divided by the fast mode runtime.

$$vs(c,p,i,m) = \frac{r_{\text{novec}}(c,p,i,m)}{r_{\text{fast}}(c,p,i,m)} \tag{4.9}$$

The vectorization speedup stability gives users the idea how far on average the speedup is close to the largest speedup in the same mutation group. Formally, the vectorization speedup stability is defined as follows.

$$\mathbf{VS}_{\text{instance}}(c,p,i) = \{vs(c,p,i,m) \mid m \in \mathbf{M}(i)\} \tag{4.10}$$

$$vs^{\text{max}}_{\text{instance}}(c,p,i) = max(\mathbf{VS}_{\text{instance}}(c,p,i)) \tag{4.11}$$

$$vs^{\text{scaled}}(c,p,i,m) = \frac{vs(c,p,i,m)}{vs^{\text{max}}_{\text{instance}}(c,p,i)} \tag{4.12}$$

$$vs^{\text{stability}}_{\text{instance}}(c,p,i) = geomean(\{vs^{\text{scaled}}(c,p,i,m) \mid m \in \mathbf{M}(i)\}) \tag{4.13}$$

$$vs^{\text{stability}}_{\text{pattern}}(c, p) = geomean(\{vs^{\text{stability}}_{\text{instance}}(c, p, i) \mid i \in \mathbf{I}(p)\}) \tag{4.14}$$

$$vs^{\text{stability}}(c) = geomean(\{vs^{\text{stability}}_{\text{pattern}}(c, p) \mid p \in \mathbf{P}\}) \tag{4.15}$$

The description for each equation is similar to that of the runtime stability. Note, however, that since we use the max speedup as a reference point, it is the denominator in equation 4.12, in order to make the scaled vector speedup fall between 0 and 1..

As opposed to runtime stability, the highest speedup alone does not indicate that it is the fastest execution time. Higher vector speedup stability does not always mean the compiler is doing a good job with vectorization. It only means the compiler is stable with regards to vector speedup. Consider the example in table 4.4, where m1 and m2 are two mutations of the same instance.

	Mutation m1			Mutation m2		
	r_{novec}	r_{fast}	vs	r_{novec}	r_{fast}	vs
Compiler c1	24	6	4	24	8	3
Compiler c2	40	10	4	30	10	3
Compiler c3	80	20	4	20	10	2
Compiler c4	10	20	0.5	10	20	0.5

Table 4.4: Higher speedup is not always the desired speedup

For compiler c1, a higher speedup in m1 results from a faster runtime in vector mode. This is the only scenario where the most beneficial speedup is the maximum speedup in the mutation group. It leads to the fastest runtime. On the other hand, for compiler c2, a higher speedup in m1 results from a slower runtime in no-vec mode. For compiler c3, although there is a higher speedup for m1, it is not a favorable mutation, since the end result we would like to achieve for a stable compiler is to have the runtime at least equal to m2 in fast mode. Finally, compiler c4 shows stability in the vector speedup. Both mutations have the same speedup. However, turning on vectorization actually slows down the program for both mutations.

Despite the caveats, it is expected that stability in vectorization capabilities will be a useful metric. When compilers are stable in novec mode, as in the case for compiler c1, vector speedup stability will correctly indicate the desired vector speedup. While for other cases this metric may not always reflect how far the compiler is from the most beneficial vectorization performance, high variation in vectorization speedup will still result in low vector stability, which will inform the user about potential performance bugs.

Each metrics is useful for a certain aspect. To get a general idea of the overall compiler's

performance, compiler developers should look at all the metrics presented.

4.2.5 Cost Model Stability

Vectorization does not always lead to speedup. Modern compilers use cost models during compilation to estimate the performance gains if vectorization is applied. If the cost model determines that vectorization is not profitable, it does not vectorize the code. A perfect compiler with a perfect cost model will always do the right thing, which means that enabling the cost model will never slow down the program. The cost model stability tells users the average speedup gained from turning on the cost model.

The definition of cost model speedup mostly mirrors that of vector speedup. The only difference is that it compares r_{fast} to $r_{\text{nopredict}}$, as opposed to r_{novec}.

$$cm(c,p,i,m) = \frac{r_{\text{nopredict}}(c,p,i,m)}{r_{\text{fast}}(c,p,i,m)} \tag{4.16}$$

$$\mathbf{CM}_{\text{instance}}(c,p,i) = \{cm(c,p,i,m) \mid m \in \mathbf{M}(i)\} \tag{4.17}$$

$$cm^{\text{max}}_{\text{instance}}(c,p,i) = max(\mathbf{CM}_{\text{instance}}(c,p,i)) \tag{4.18}$$

$$cm^{\text{scaled}}(c,p,i,m) = \frac{cm(c,p,i,m)}{cm^{\text{max}}_{\text{instance}}(c,p,i)} \tag{4.19}$$

$$cm^{\text{stability}}_{\text{instance}}(c,p,i) = geomean(\{cm^{\text{scaled}}(c,p,i,m) \mid m \in \mathbf{M}(i)\}) \tag{4.20}$$

$$cm^{\text{stability}}_{\text{pattern}}(c,p) = geomean(\{cm^{\text{stability}}_{\text{instance}}(c,p,i) \mid i \in \mathbf{I}(p)\}) \tag{4.21}$$

$$cm^{\text{stability}}(c) = geomean(\{cm^{\text{stability}}_{\text{pattern}}(c,p) \mid p \in \mathbf{P}\}) \tag{4.22}$$

The cost model stability has the same caveats when interpreting results as the vector speedup stability.

4.2.6 Peer Comparison Metrics

The previously defined metrics focus on stability. Given a single compiler, it indicates how much variability is there with different mutations in the same mutation group and potentially how far it is from the best performance. While a compiler may be highly stable, it does not mean that it produces fast code. We believe it is also helpful for users to see comparative performance between different compilers.

The peer comparison metrics include top rank proportion, bottom rank proportion, better rank proportion, and peer speedup.

When comparing runtimes, we account for noise. It is unlikely that comparable runtimes

will be exactly equal at the nanosecond level. We consider two runtimes to approximate each other when they are within 5% runtime of each other. For one runtime to be considered faster than another, we should not include the cases where they approximate each other. We define the $\approx$ operator and the $<<$ operators as follows.

$$r_1 \approx r_2 = max(r_1, r_2) \leq 1.05 \times min(r_1, r_2) \tag{4.23}$$

$$r_1 \not\approx r_2 = \neg(r_1 \approx r_2) \tag{4.24}$$

$$r_1 \ll r_2 = r_1 < r_2 \wedge r_1 \not\approx r_2 \tag{4.25}$$

We also define a helper function that converts booleans to either 0 or 1 to simplify the definitions of some of the metrics.

$$count(b) = \begin{cases} 1 & \text{if } b = true \\ 0 & \text{otherwise} \end{cases} \tag{4.26}$$

With the function *count*, we can calculate the proportion of 1s in a multiset whose values are 0 or 1 by calculating the arithmetic mean of that multiset.

4.2.7 Top Rank Proportion

Top rank proportion tells the user on average, how often a compiler performs better than the rest of the compilers in the experiment. The top rank proportion is formally defined as follows.

$$\mathbf{R}_{\text{compilers}}(p, i, m) = \{r_{\text{fast}}(c, p, i, m) \mid c \in \mathbf{Compilers}\} \tag{4.27}$$

$$r^{\text{min}}_{\text{compilers}}(p, i, m) = min(\mathbf{R}_{\text{compilers}}(p, i, m)) \tag{4.28}$$

$$isTop(c, p, i, m) = r_{\text{fast}}(c, p, i, m) \approx r^{\text{min}}_{\text{compilers}}(p, i, m) \tag{4.29}$$

$$top^{\text{proportion}}_{\text{instance}}(c, p, i) = mean(\{count(isTop(c, p, i, m)) \mid m \in \mathbf{M}(i)\}) \tag{4.30}$$

$$top^{\text{proportion}}_{\text{pattern}}(c, p) = mean(\{top^{\text{proportion}}_{\text{instance}}(c, p, i) \mid i \in \mathbf{I}(p)\}) \tag{4.31}$$

$$top^{\text{proportion}}(c) = mean(\{top^{\text{proportion}}_{\text{pattern}}(c, p) \mid p \in \mathbf{P}\}) \tag{4.32}$$

First, equation 4.27 groups the runtimes for the same mutation but different compilers together. Next, equation 4.28 finds the best (minimum) runtime across compilers for each mutation. Equation 4.29 defines a function to determine whether a compiler is a top compiler for a given mutation. Here we allow noise and approximate runtimes. Any compiler whose mutation runtime approximates the top runtime is still considered a top compiler for that

mutation. With this definition, it is possible to have multiple top compilers for a mutation. Equation 4.30 calculates the proportion of mutations for a given instance where a compiler performs the best. Equation 4.31 averages this proportion at the pattern level. Finally, equation 4.32 averages this proportion across all patterns.

4.2.8 Bottom Rank Proportion

Compiler developers may want to know when their compilers perform poorly. Therefore we believe that the bottom rank proportion would be as useful as the top rank proportion. The definition of the bottom rank proportion is almost the same as the top rank. Note again that when a runtime approximates the worst runtime, it is also considered the worst runtime.

$$r^{\text{max}}_{\text{compilers}}(p, i, m) = max(\mathbf{R}_{\text{compilers}}(p, i, m)) \tag{4.33}$$

$$isBottom(c, p, i, m) = r_{\text{fast}}(c, p, i, m) \approx r^{\text{max}}_{\text{compilers}}(p, i, m) \tag{4.34}$$

$$bottom^{\text{proportion}}_{\text{instance}}(c, p, i) = mean(\{count(isBottom(c, p, i, m)) \mid m \in \mathbf{M}(I))\}) \tag{4.35}$$

$$bottom^{\text{proportion}}_{\text{pattern}}(c, p) = main(\{bottom^{\text{proportion}}_{\text{instance}}(c, p, i) \mid i \in \mathbf{I}(p)\}) \tag{4.36}$$

$$bottom^{\text{proportion}}(c) = mean(\{bottom^{\text{proportion}}_{\text{pattern}}(c, p) \mid p \in \mathbf{P}\}) \tag{4.37}$$

4.2.9 Better Rank Proportion

Users may want a closer look at the compiler comparison. We define the peer rank proportion to capture this. The better rank proportion measures how frequently one compiler performs better than another.

$$isBetter(c_1, c_2, p, i, m) = r_{\text{fast}}(c_1, p, i, m) \ll r_{\text{fast}}(c_2, p, i, m) \tag{4.38}$$

$$better^{\text{proportion}}_{\text{instance}}(c_1, c_2, p, i) = mean(\{count(isBetter(c_1, c_2, p, i, m)) \mid m \in \mathbf{M}(i)\}) \tag{4.39}$$

$$better^{\text{proportion}}_{\text{pattern}}(c_1, c_2, p) = mean(\{better^{\text{proportion}}_{\text{instance}}(c_1, c_2, p, i) \mid i \in \mathbf{I}(p)\}) \tag{4.40}$$

$$better^{\text{proportion}}(c_1, c_2) = mean(\{better^{\text{proportion}}_{\text{pattern}}(c_1, c_2, p) \mid p \in \mathbf{P}\}) \tag{4.41}$$

4.2.10 Peer Speedup

The top rank, bottom rank, and better rank proportions are all proportions of how often a compiler is better or worse than others. A user may also want to know how much better it is than its peers. The peer speedup tells the user, for cases where one compiler is better

than another, how much better is it. This value will always be larger than 1 since it only takes into account the cases where a compiler performs better than its peers.

$$ps(c_1, c_2, p, i, m) = \frac{r_{\text{fast}}(c_2, p, i, m)}{r_{\text{fast}}(c_1, p, i, m)} \tag{4.42}$$

$$\mathbf{PS}_{\text{instance}}(c_1, c_2, p, i) = \{ps(c_1, c_2, p, i, m) \mid m \in \mathbf{M}(i) \land \mathit{isBetter}(c_1, c_2, p, i, m)\} \tag{4.43}$$

$$ps_{\text{instance}}(c_1, c_2, p, i) = \mathit{geomean}(\mathbf{PS}_{\text{instance}}(c_1, c_2, p, i)) \tag{4.44}$$

$$ps_{\text{pattern}}(c_1, c_2, p) = \mathit{geomean}(\{ps_{\text{instance}}(c_1, c_2, p, i) \mid i \in \mathbf{I}(p)\}) \tag{4.45}$$

$$ps(c_1, c_2) = \mathit{geomean}(\{ps_{\text{pattern}}(c_1, c_2, p) \mid p \in \mathbf{P}\}) \tag{4.46}$$

4.2.11 Alternative Measurements Though Hardware Counters

All of the metrics we have discussed are based on using the runtime of programs. An alternative method for measuring performance is through hardware counters. Hardware counters can be useful and give users more detailed information about performance such as floating-point operations per second (FLOPS), vector instruction rates, or a roofline plot [61].

The main drawback for this in our context is that, in order to reliably measure hardware counters, each program would have to run for a long enough time. In our preliminary experiments with measuring vector instruction rates, some programs require up to 20 seconds for the Intel VTune Amplifier (version 2019 Update 4 build 597835) to consistently and reliably return the vector instruction rates.

With today's technology, the amount of time needed to measure vector instruction rates would be too long for our purposes. Our goal is to generate a large number of tests to find compiler bugs. Since performance tests can not be run in parallel, the total time for an experiment would take too long. For example, with our machine configuration, for 1000 programs for one compiler with 3 compilation modes would take approximately at least 16 hours to capture all the vector instruction rates. On the other hand, when we only measure runtimes, each program typically runs in less than one second, and the same experiment setup would take less than an hour to complete. We were able to iterate much more quickly by relying only on runtimes.

While we may miss opportunities to get a more detailed perspective on performance by using hardware counters, runtimes are straightforward and easy to understand. We opted for a simpler approach during the exploratory stage, and it became a reasonable choice for our metrics.

It is possible that it would take shorter to reliably measure other hardware counters than the ones related to vector instruction rate. We are also aware of tools such as the

Modular Assembly Quality Analyzer and Optimizer (MAQAO [62]) that can approximate some hardware counts statically. However, we did not explore this direction further. The runtime served the thesis's purpose. A reliable and fast way to measure hardware counters would complement, not replace, runtimes.

4.3 STATISTICAL QUANTIFICATION OF RESULTS

Since our approach is based on randomness, we must be careful in claiming any results. In particular, any source of random noise should be quantified. Many of our results can be quantified through statistical methods. In this section, we discuss relevant methods that are used in our study.

4.3.1 Inferential and Descriptive Statistics

We make use of both the broad categories in statistics, namely inferential and descriptive statistics. Each of our experiments result in sample data. We can make claims about the samples in two different ways.

When we describe properties of a specific sample, we are using descriptive statistics. It gives users information about that particular sample and nothing more. An example of how we use descriptive statistics is when reporting outliers of our random sample. It describes data points, such as the min, max, or average, that are specific to the sample.

On the other hand, if we make broader claims about the general population that a sample is drawn from, we are using inferential statistics. An example of inferential statistics is reporting results using confidence intervals. Rather than report the average speedup for a given sample, using inferential statistics we can claim that we are 95% confident that the average speedup of the population, not just the sample, lies within some given interval. In order to make such claims, certain requirements about the data and sampling method must be met. We discuss such requirements later in this section.

4.3.2 Using the Arithmetic Mean or Geometric Mean Properly

We have used the arithmetic mean and geometric mean in our metrics. Here we discuss their differences and the proper situations to use each. Any new metric added to the system should take into account these distinctions.

The arithmetic mean should be used where additive operations are the natural way of accumulation. For example, the arithmetic mean should be used when the multisets $\{1, 3\}$

and $\{2, 2\}$ are considered to have the same average, which is 2. This is meaningful for raw numbers such as the runtimes of each iteration of the program. Another place where it is meaningful in our experiments is for counting, such as when calculating proportions in the top rank, bottom rank, and better rank metrics.

The geometric mean should be used where multiplicative operations is the natural way of accumulation. For example, the geometric mean should be used when the multisets $\{1, 4\}$ and $\{2, 2\}$ are considered to have the same average, which is 2. This is meaningful especially for speedups, when half the speedup and twice the speedup averages to no speedup. It is meaningful when used for calculating average speedups as in the runtime stability, vector speedup stability, cost model stability, and peer speedup stability. One caveat when using the geometric mean is that the values cannot contain zeros. Fortunately, speedups are never zero, since that would mean that one program runs infinitely faster than the other.

While the arithmetic mean is straightforward and intuitive, the geometric mean is less intuitive and has caused controversy in the literature that readers should be aware of. One main benefit of using the geometric mean is that it preserves the ordering of evaluated compilers after normalization [63]. Speedup is a form of normalization since it "normalizes" the runtime against a reference runtime. The arithmetic mean may differ depending on which compiler we use as a reference, while the geometric mean is consistent. On the other hand, the geometric mean throws away information regarding the total runtime [64], meaning that tests that run very long and tests that run very short have the same weight.

Both claims are valid. Mashey [65] argues that one must further determine the type of analysis being performed to make the correct decision on the type of mean to use. Using the geometric mean discards information about the total runtime, an important property when analyzing workloads. Workloads not only represent the types of programs under test, but also their frequency. On the other hand, when the main analysis is on the relative performance of programs, where workload is not a concern, the geometric mean is still a suitable method [66]. Our work focuses on the latter, which justifies the use of the geometric mean. The reader, however, must be aware of these nuances and make the correct modifications to the system if the underlying premises change.

4.3.3 Confidence Intervals

When making inferences about the population through a sample, one method used to report results are confidence intervals. The confidence interval tells the reader a range of possible values, and quantifies the sampling process with a probability number. For example, if we report from a random sample of 100 patterns that the speedup of compiler c_1

over compiler c_2 lies in $[1.5, 1.75]$ with confidence level 95%, it means that when we repeat the process of sampling and computing the speedup range, the range will include the true population speedup 95% of the time. Note that the confidence level quantifies the process, and not the interval [67]. It may also be helpful for the reader to read this result as, "we are 95% confident that our sampling process includes the true speedup range, and we calculated the range to be [1.5, 1.75]".

Being able to quantify results as a range with probability is useful, especially when reproducibility of the results is important. In particular, when we report a confidence interval and somebody repeats the experiment with a new random sample, their confidence interval will very likely overlap with ours, since both our and their intervals have a 95% change of covering the true value. For our experiments, we report some results on reproducibility in sections 4.9.

Confidence intervals work because of a result in statistics called the central limit theorem. The theorem states that when we sample from a population, where each sample is independent and identically distributed (i.i.d.), if the sample size is large enough, the sampling distribution of the arithmetic mean will approximate a normal distribution.

This result can be applied to our context. Suppose an experiment consists of randomly generating a large number of tests and computing their arithmetic mean. When we generate the tests the same way and each test is independent from one another, then our sample is i.i.d. When we repeat this process, computing the arithmetic mean of new random samples over and over, it results in a distribution called the sampling distribution of the mean. The central limit theorem states that the sampling distribution of the mean will approximate a normal distribution.

When we know that a distribution is normal, it is possible to determine the range of values for a given probability. For example, when we know that the distribution of mean compiler speedups is normal, we know which range of speedups take up 95% probability by looking up a probability density function table for normal distributions.

Fortunately, our metrics all rely on either the arithmetic mean or geometric mean. For the arithmetic mean, it is straightforward to calculate the confidence interval. For a sample $X = \{x_1, ..., x_N\}$, which is a set of N values, the 95% confidence interval ci_{95} is calculated through the following formula [45].

$$std(X) = \sqrt{\frac{\sum_{i=1}^{N}(x_i - mean(X))^2}{N-1}} \tag{4.47}$$

$$ci_{95}(X) = mean(X) \pm \mathrm{t}_{(0.025,N-1)}\frac{std(X)}{\sqrt{N}} \tag{4.48}$$

The function std calculates the sample standard deviation. The value of t is a table look up to determine how to scale the standard deviation properly in an approximately normal distribution, so that it determines the intervals. In our experiment, we are interested in the 95% confidence interval for a sample with size N. For our purposes, we are interested in a symmetric distance surrounding the mean. In statistics, this is called a two-tailed test, and the significant level for 95% confidence is $(1-95)/2 = 0.025$. Another concept is the degree of freedom. For estimating the mean, this number is always $N-1$. Therefore, to compute the 95% confidence interval, we would look up $\mathrm{t}_{(0.025,N-1)}$ in our formula.

The confidence interval for the geometric mean needs additional handling, since the main result of the central limit theorem applies only to the arithmetic mean. We can work with geometric means by transforming data to a log scale, calculating the confidence interval, then transform the intervals back at the end [68]. Notice the following equation.

$$\mathbf{Log}(X) = \{log(x) \mid x \in X\} \tag{4.49}$$

$$geomean(X) = exp(mean(\mathbf{log}(X))) \tag{4.50}$$

Subsequently, we can compute the confidence intervals for the geometric mean ci_{95}^{geo} as follows.

$$ci_{95}(log(X)) = [a, b] \tag{4.51}$$

$$ci_{95}^{geo}(X) = [exp(a), exp(b)] \tag{4.52}$$

4.3.4 Checking Data Requirements

For the central limit theorem to apply, we must ensure that the sampling distribution for our metrics approximate a normal distribution, and that each data point in a sample is i.i.d. For each metrics, we need to ensure these conditions hold.

One of our preliminary experiments (not included in this thesis) consisted of 10 patterns, with 10 instances per pattern, and 10 mutations per instance. Since this setup included $10 \times 10 \times 10 = 1000$ tests, we assumed it was large enough to report valid confidence intervals. However, when we repeated the experiment, we found that the results deviated from the original experiment. This is a result from failing to check the data requirements before reporting confidence intervals.

It is important to ensure that the requirements of the central limit theorem is met in order for a reported confidence interval to be valid. In particular, we are able to use the T distribution table look up since we assume the sampling distribution of the mean to be

normal. It is also shown by the theorem that for a sufficiently large sample size, the sampling distribution of the mean approximates the normal distribution. Our job is then to determine what is a large enough sample size.

Practically, to determine whether a sample size is large enough, we would experiment with various sample sizes to obtain different sampling distribution. We can then perform statistical tests to check which sampling distribution starts to approximate the normal distribution. When a large enough sample size induces a sampling distribution approximating the normal distribution, it means that any sample size with at least that size meets the requirement.

The other requirement is that our data is i.i.d. We must show that each data point in the sample is independent and is identically distributed. We describe how we meet these requirements below.

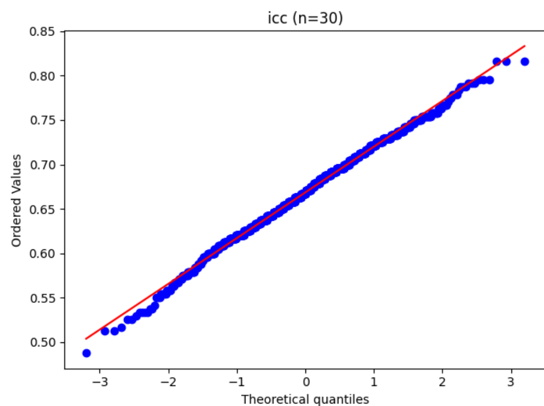
First, each data point we use in our sample must be independent. Since mutations are generated from instances, and instances are generated from patterns, there is a hierarchy in our data. In particular, mutations in the same mutation group are related, and instances that belong to the same pattern are also related. Only patterns are independent in our model, and any data point in the sample should be at the pattern level. For this reason, we treat patterns as a whole and sample data at the pattern level. Whatever instance and mutation that belongs with a sampled pattern must be included as a whole. This is the main reason that for each metric, we gather results at the pattern level before aggregating all the patterns.

The second requirement for i.i.d. is that each data point must come from identical distributions or in other words, the same population. This is important for reproducible results. Researchers who wish to replicate our experiments should sample each data point from the same population that we sample from. Not only does this mean they should use the same pattern, instance, and mutation profiles, but they also need to ensure that they sample data the same way as us. Suppose our experiment includes 200 patterns, with 2 instances per pattern and 4 mutations per instance. Essentially, each data point in our population are "patterns with 2 instances per pattern and 4 mutations per instance". While any research who wishes to reproduce our results may sample different number of patterns, the instance per pattern and mutation per instance should match ours to meet this requirement of computing confidence intervals.

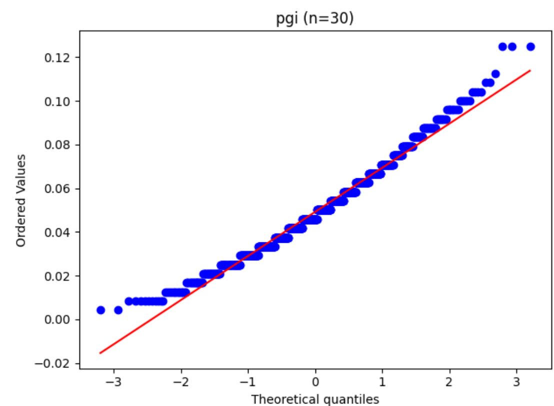
Let us consider the top rank proportion metric as an example. To meet the i.i.d. requirement, each data point is measured at the pattern level, which is $top^{\text{proportion}}_{\text{pattern}}$. We aggregate this data with the arithmetic mean to get the top rank proportion $top^{\text{proportion}}$. In order for us to report any results with the confidence interval, we need to check whether the sampling distribution of $top^{\text{proportion}}$ approximates a normal distribution or not. We use a

quantile-quantile (QQ) plot [69] to check this property. A QQ plot is a scatter plot that plots the quantiles of two distributions against each other. In our case, we would like to see how the sampling distribution of $top^{\text{proportion}}$ pairs up with the normal distribution. If the data points approximate a straight diagonal line and gather around the middle, with some outliers spread equally at both ends, then $top^{\text{proportion}}$ approximates a normal distribution.

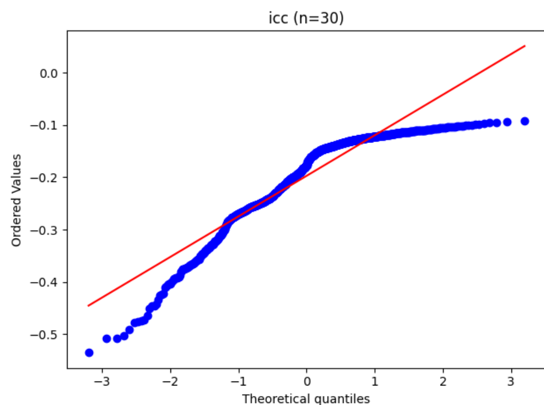
Various variables affect the distribution normality of our metrics. This can be illustrated through different QQ plots we have explored while determining the sample size for our experiments. A typically sample size recommended by statisticians is 30 for the sampling distribution to approximate normalilty. This can be seen in figure 4.1a, which shows the sampling distribution for $top^{\text{proportion}}$ stability for ICC with sample size 30 and with loop unroll as the method for generating mutations.



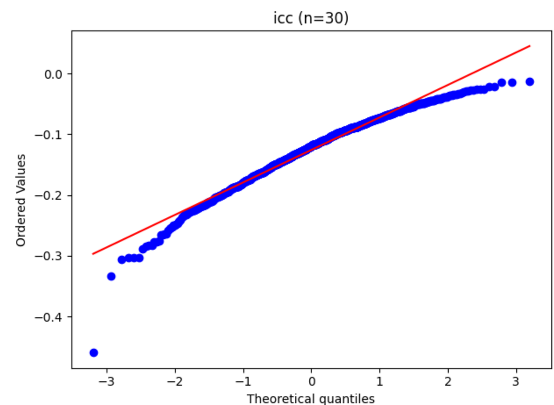
(a) Against $top^{\text{proportion}}$ for ICC where loop unroll is used to generate mutations



(b) Against $top^{\text{proportion}}$ for PGI where loop unroll is used to generate mutations



(c) Against $r^{\text{stability}}$ for ICC where loop unroll is used to generate mutations



(d) Against $r^{\text{stability}}$ for ICC where loop interchange is used to generate mutations

Figure 4.1: QQ plots with a normal distribution X-axis for sample size 30

However, for PGI (figure 4.1b), the distribution is not normal, even for the same metric,

same sample size, and same mutation generation process.

Even for ICC, with the same sample size and same mutation generation process, we can have a sampling distribution that is not normal when we consider a different metric, as shown in figure 4.1c.

Finally, the mutation generation process also affects the sampling distribution. Figure 4.1c and figure 4.1d both show the QQ plot for the sampling distribution of the runtime stability for ICC with sample size 30. The only difference is that figure 4.1d uses loop interchange to generate mutations, resulting in a sampling distribution slightly more normal than using loop unroll.

For this thesis, we determined that the sample must include at least 100 patterns for all the compilers, all the transformations, and all the metrics to have a sampling distribution of the metrics approximate a normal distribution. Our experiments included 200 patterns to handle unexpected results that may cause the sampling distribution to deviate from normal.

It is important to check for normality when a compiler, a transformation, or a new metric is added. Otherwise, reporting the confidence intervals may not include valid results.

4.3.5 Alternative Methods for Checking Normality

The QQ plot is a good visual indicator to check for the normality of a distribution. There are other statistical checks that can be done quantitatively to measure how close a distribution is to a normal distribution. These include, but are not limited to, the Kolmogorov-Smirnov (KS) and Shapiro-Wilk tests [70]. We did not study these tests any further when checking for normality. The QQ plots were analyzed by hand. In order to automate the process of checking the normality requirements of any metric, any compiler, and any sample size, we would likely need to incorporate these.

4.3.6 Reporting Confidence Intervals Without the Normality Assumption

The distribution we observe could be non-normal. This can be because of the data, the metric, or both. It also could be that we do not have enough number of data points in our sample. In this case, a method called bootstrapping [71] can be used. Originally, we experimented with this method. After we were able to satisfy the data requirements for normality, we did not pursue bootstrapping further.

4.4 COMPILER EVALUATION: LOOP UNROLL

In this section we describe the first of three experiments studying compiler performance with our test generation system. The generated sample in this study include 200 patterns, with 2 instances per pattern, and 4 mutations per instance, totaling 1600 tests. The pattern and instance profiles for this study are shown in the table 4.5a and 4.5b, respectively.

Property	Value
Array	{A(1), B(1), C(1), D(1), E(1)}
Coeff	$\{a_1, a_2, a_3\}$
Const	$\{b_1, b_2, b_3\}$
Data	$\{f_1, f_2, f_3\}$
LoopVar	$\{i_1, i_2, i_3\}$
# Loops	1
# Stmts	3
# Ops	4
LoopDepth	1
Ops	$\{+, *, -\}$

(a) Pattern profile

Set	Range of values
Coeff	[0, 2]
Const	[0, 16]
Data	[0.5, 2.0]
$\text{LoopVar}_{\geq}$	[1000000, 2000000]
$\text{LoopVar}_{\leq}$	[3000000, 4000000]

(b) Instance profile

Table 4.5: Profiles for loop unroll experiment

In this study, we use loop unroll to generate mutations. Loop unroll is a compiler transformation that transforms the loop body so that it contains more than one iteration of the original loop body, so that while the transformed loop body is larger, the loop has fewer iterations. One of the benefits of loop unrolling is to reduce the number of times a loop test needs to be performed. The unrolling factor is chosen randomly between 1 and 16, with 1 meaning not unrolling the loop at all. Loop unroll is always legal, as long as the number of iterations is larger than the unroll factor. Therefore there are 16 possible mutations for each instance. We make sure there are no duplicates in each mutation group.

Among all the patterns, 10 included tests that were excluded from our results. One of the excluded patterns included a reduction. ICC and GCC both had cases where results differed 2% from the median checksum, when our threshold is 1%. Another failed pattern was due to an ICC internal error during compilation. This only happened to the mutation that unrolled the loop 4 times. The other mutations of the same instance unrolled 5, 10, and 15 times and there was no internal error during compilation. The rest of the excluded patterns were caused because the checksum calculation overflowed, resulting in either inf or -inf. The 190 remaining patterns were correct.

Stability results are shown in figure 4.2. The error bars represent 95% convidence intervals for the metrics.

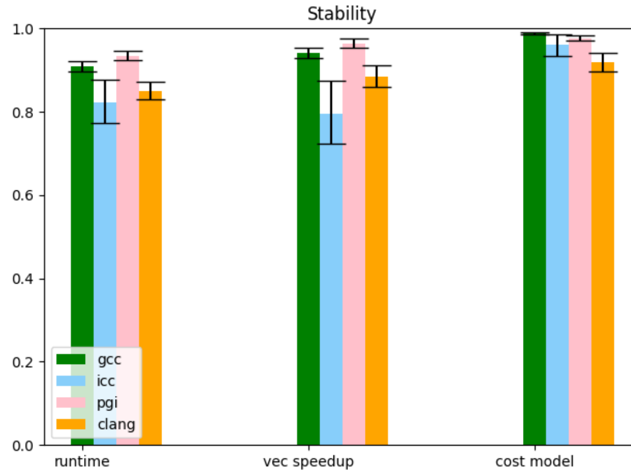


Figure 4.2: Stability results for loop unroll study

In general, PGI was more stable than the other three compilers. ICC was the least stable, and has a larger variation of runtimes and vector speedups as seen by the wider confidence interval. Overall, GCC has higher stability than Clang.

Figures 4.4, 4.5, and 4.6, show the distribution of the scaled runtimes, vector speedups, and cost model speedups. The green dots represent 90% of the data points, while the red dots represent the 10% outliers.

For all of the compilers, the majority of data points have acceptable scaled runtimes, vector speedups, and cost model speedups.

While the majority of tests have reasonable scaled runtimes, vector speedups, and cost model speedups, the outliers are extreme. For example, the lowest scaled runtime for ICC is x0.000005. For that instance, unrolling the loop with factor 2 was approximately 180,000 times faster than unrolling it with factor 5, while it was 600 times faster than unrolling it with factor 3. The outlier code is shown in figure 4.3.

```
// output dependence only
for (int i1 ...) {
    A[0 * i1 - 0]  = ...;
    B[0 * i1 + 13] = ...;
    D[0 * i1 - 0]  = ...;
}
```

Figure 4.3: ICC's runtime stability outlier for unroll study

The code does not need to iterate through the whole iteration space of loop i1, since for this instance, the coefficient of the loop index is zero, making all the array indices constant.

The assembly output of the compiler indicates that, different unrolling factors prevented ICC from removing deadcode caused by the repetitive assignment to the same locations and resulted in code iterating through the loops.

Figures 4.7, 4.8, and 4.9 show the top and bottom rank, better rank, and peer speedup metrics, respectively.

While ICC was one of the less stable compilers with regards to handling loop unroll, it produced the fastest code most frequently and slowest code least frequently, as shown in figure 4.7. PGI, on the other hand, while being the most stable, produces the slowest code most of the time, as shown in the same figure. We hypothesize that when a compiler produces fast code, small variations affect the stability more than when compared to slow code. While GCC and Clang produce the fastest code with comparable frequency, Clang produces slow code more often.

Next, figure 4.8 compares compilers in pairs. It is clear that ICC performs better than the other compilers. When comparing GCC and Clang directly, Clang produces faster code more often. Finally, PGI rarely produces faster code than other compilers.

Figure 4.9 shows the peer speedup for all compilers. Recall that peer speedup only takes into account the case when a compiler is faster than another, thus always having values higher than 1. The labels on the x-axis specify which pair of compilers are being measured. For example, the label GCC/ICC shows the average speedup when ICC is faster. (To compute this speedup, the runtime of GCC is divided by the runtime of ICC.)

Notice that some confidence interval bars are missing. Recall in Section 4.3.4 that for our study, we need at least 100 data points in order to report a valid confidence interval. Since the peer speedup metric only includes cases that one compiler is faster than another, a compiler that performs worse most of the time would not have enough data points to form a confidence interval.

Overall, one compiler on average does not have a speedup more than x2 when compared to another compiler. ICC not only has the most top rank, but when it performs better than other compilers, it also has a decent speedup, no less than x1.25 relative to the other three compilers. The pair GCC and Clang are, again, comparable, with overlapping confidence intervals. However, when Clang is faster, there is more variation in speedup as seen by the wider confidence interval. PGI does not have enough data points for us to make claims about the population of all tests. It can be only said that for this sample, when it performs well, we see speedups up to more than x1.50.

Finally, figures 4.10, 4.11, 4.12, and 4.13 show the slowdown distribution of of compiler pairs. We show the slowdown distribution, as opposed to the speedup distribution, since slowdowns always have values between 0 and 1. This is consistent with other plots and

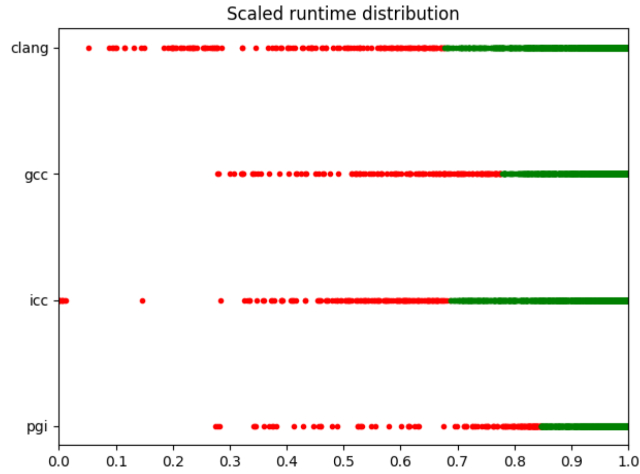


Figure 4.4: Scaled runtime distribution of loop unroll study

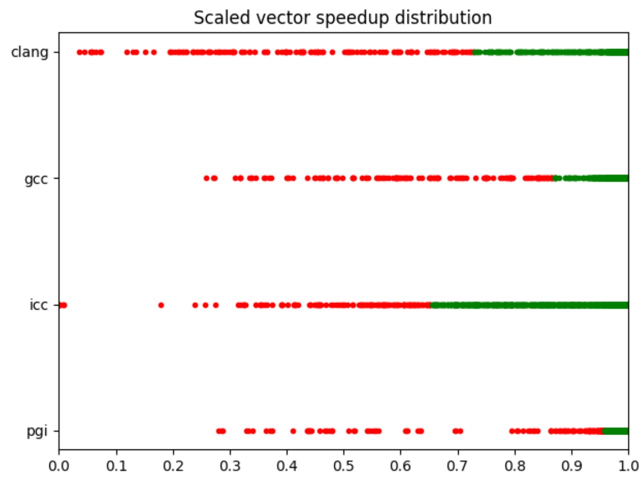


Figure 4.5: Scaled vector speedup distribution of loop unroll study

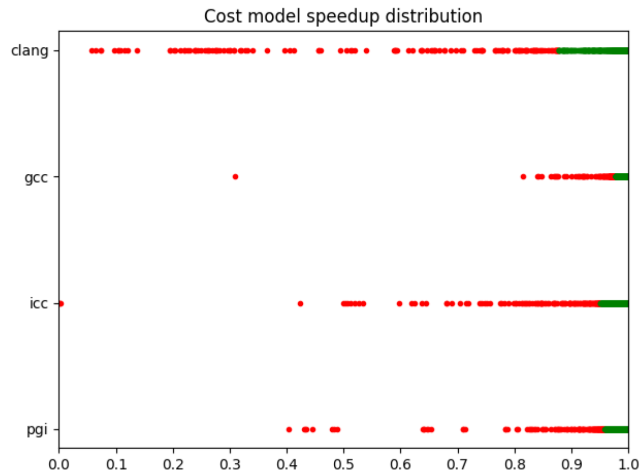


Figure 4.6: Scaled cost model speedup distribution for loop unroll study

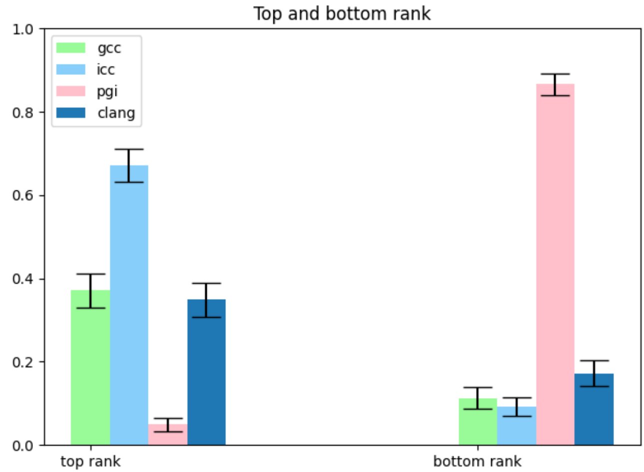


Figure 4.7: Top and bottom rank proportions for loop unroll study

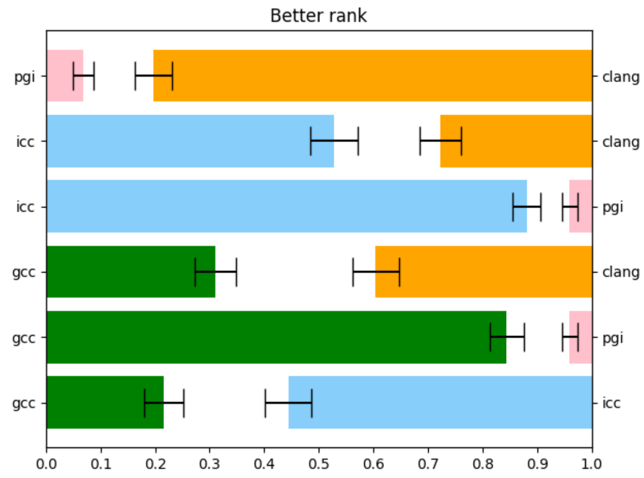


Figure 4.8: Better rank proportion for loop unroll study

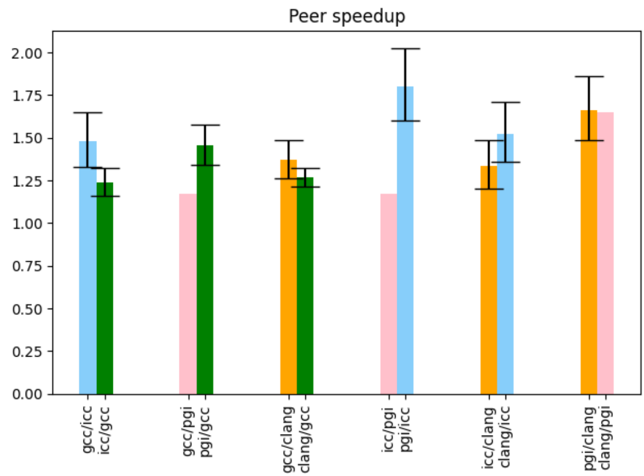


Figure 4.9: Peer speedup for loop unroll study

Figure 4.10: Clang's slowdown distribution for unroll study

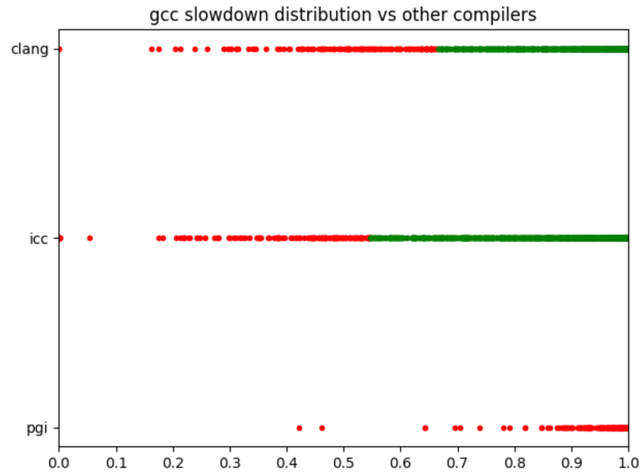


Figure 4.11: GCC's slowdown distribution for unroll study

avoids scaling issues when the outliers are extreme. For example, for some tests, ICC's compiled program ran up to x740000 faster than PGI's. When scaling this graph, it would be difficult to see the distribution of data points if other distributions in the same plot have a different scale. We do not throw away any information by showing only the slowdown, since the slowdown is just a reciprocal of the speedup.

We must point out that for these figures, the red dots represent the bottom 10% of all the slowdown and speedups, not just the slowdowns. This means that it is possible for a plot to include only red dots if the compiler does well in the majority of the data points, and the slowdowns only include outliers. An example of this is when GCC is compared to PGI in figure 4.11. The plot includes only outliers, since the 90% majority of the data points are speedups.

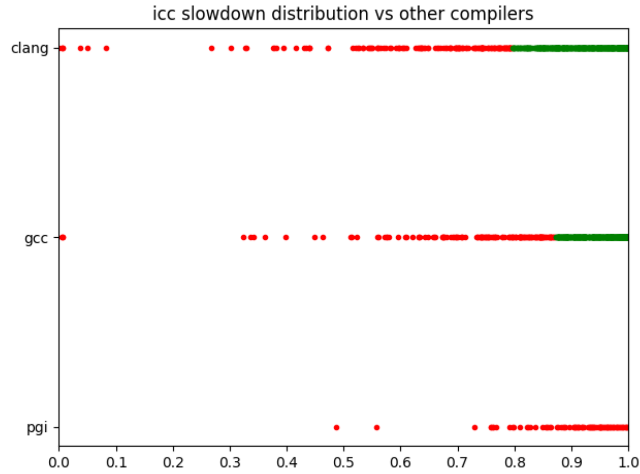


Figure 4.12: ICC's slowdown distribution for unroll study

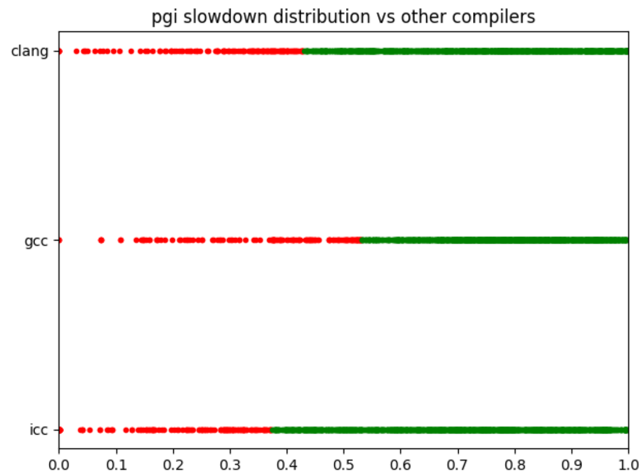


Figure 4.13: PGI's slowdown distribution for unroll study

Similarly to the stability metrics, outliers can be extreme, even for the best-performing compiler as was observed above regarding ICC. For example, for one whole mutation group, Clang had a speedup over ICC as high as x140000. When we looked at these extreme cases further, we found that they are similar to the outlier for runtime stability (figure 4.3). In particular, they contain computations that make repetitive changes to a fixed array element and contain no reductions. The ability of different compilers to detect this behavior varies. Other outliers exist that do not fall into this behavior, such as an outlier where GCC produces code that is x3 faster than Clang. We did not study all of the outliers, which include the bottom 10%, totaling 160 mutations for each metric.

4.5 ON THE SPACE OF POSSIBLE PATTERNS

This section discusses the calculations for approximating the space to give the reader a sense of the size of the space of possible patterns that can be generated from a pattern profile.

Consider the pattern profile in Table 4.5a. There are 3 assignments. The left hand side of the assignment must be an array. There are 5 choices, totaling $5^3 = 125$ ways to fill the left hand sides.

On the right hand side there are 4 operations. Since all operators are binary, there are 5 operands. An operand can be either an array or data, so there are $5 + 3 = 8$ choices. For 3 assignments and 5 operands each, there are $8^{(5*3)} \approx 3.5e13$ ways to fill in the operands on the right hand side.

For each array dimension, we have an affine function of the loop variable. Since for the pattern profile from Table 4.5a, we generate singly loop nests, only one loop variable will be present. For each affine index, there are 3 possible coefficients and 3 possible constant terms, totaling $3 * 3 = 9$ possible affine indices. For the left hand side of 3 statements, there are 3 indices to generate, totaling $9^3 = 729$ possible ways to generate all the left hand sides. There are 3 loop variables to choose from, but note that we would only pick one loop variable for this pattern profile and no matter which loop variable we pick, the resulting pattern would be equivalent.

On the right hand side of the 3 statements, there are expected to be $5/8 * (5 * 3) \approx 9$ operands that are array accesses (as opposed to literals), totaling $9^9 \approx 3.8e8$ indices to generate.

For the right hand side of each assignment, we form an expression from operator nodes. There are 4 operations, resulting in 5 possible binary trees [72]. Since there are 3 operators to choose from, for each binary tree, there are $3^4 = 81$ ways to fill the operator nodes. For each statement, there are $81 * 5 = 405$ ways to form an expression. For 3 statements, there are $405^3 \approx 6.6e7$ possible ways to create binary trees consisting 4 operations.

When putting everything together, we can approximate the upper bound of the space. The total space is $125 * 3.5e13 * 729 * 3.8e8 * 6.6e7 \approx 8e34$. This is the approximate upper bound of the space, since many of the patterns are considered equivalent. For example, if the only coefficient appearing in the pattern is a_1, then swapping all occurrences of a_1 with a_2 is still considered the same pattern and the total space would reduce in size if we take this into account.

This calculation is to give the reader a sense of the immense size of the space, making it virtually impossible to exhaustively explore the whole space even for small patterns with

restrictions on its structure. Randomly generating patterns from this space and using statistical methods, as described further in Section 4.3, is a more effective method for inferring overall performance of compilers.

4.6 INTERPRETING THE RESULTS

Before we move on to our next study, we believe it is important to emphasize the meaning of any results we report. Whatever we report, it is specific for the given population and is statistically qualified as 95% confident.

For instance, when we say that the 95% confidence interval of $top^{\text{proportion}}(icc)$ is [65, 75], we are claiming that for tests that are drawn from the population of all tests generated through the pattern profile, instance profile, and mutation profile, we are 95% confident that ICC performs better than PGI, Clang, and GCC about 65-75% of the time (as indicated by the confidence interval bar in figure 4.7). Furthermore, we are only claiming these results for the hardware we use, the compiler versions, and the compiler flags that we described.

This may sound restrictive, but this is an improvement over reporting results on a fixed set of tests. The pattern, instance, and mutation profiles describes a population of tests that cover much more ground while the statistical methods helps keep our results valid across experiments.

Another important point is that we are reporting the confidence interval for each metric and compiler separately. For example, we claim a 95% confidence interval for $r^{\text{stability}}(icc)$ and also a 95% confidence interval for $cm^{\text{stability}}(clang)$, we are not claiming that we are 95% confident that both of these values will fall under the two intervals concurrently. The chance of both falling under the reported intervals concurrently is $95 \times 95 = 90.25\%$ and shrinks exponentially with each metric and compiler added. In our reports, we are claiming we are 95% confident for these intervals when considering each metric and compiler independently.

4.7 COMPILER EVALUATION: LOOP INTERCHANGE

Our second study was loop interchange. Loop interchange permutes the ordering of a loop nest. An example benefit of loop interchange is increased data locality, especially when after the interchange, the innermost loop iterates over the innermost dimension of an array used in the computation. Loop interchange is not always valid. For it to be valid, it must not reverse any of the data dependences.

We generated 200 patterns, with 2 instances per pattern and 4 mutations per instance,

totaling 1600 tests. Our pattern and instance profiles are shown in Tables 4.6a and 4.6b, respectively.

Property	Value
Array	{A(3), B(3), C(3), D(3), E(3)}
Coeff	$\{a_1, a_2, a_3\}$
ZeroCoeff	$\{z\}$
Const	$\{b_1, b_2, b_3\}$
Data	$\{f_1, f_2, f_3\}$
LoopVar	$\{i_1, i_2, i_3\}$
# Loops	1
# Stmts	2
# Ops	1
LoopDepth	3
Ops	$\{+, *, -\}$

(a) Pattern profile

Set	Range of values
Coeff	[1, 2]
ZeroCoeff	[0, 2]
Const	[0, 16]
Data	[0.5, 2.0]
$\text{LoopVar}_{\geq}$	[100, 200]
$\text{LoopVar}_{\leq}$	[200, 300]

(b) Instance profile

Table 4.6: Profiles for loop interchange experiment

When compared to the loop unroll study, this study increases the loop depth and array dimensions to 3, but decreases the number of statements inside the loop to 2 and number of operations on the right hand side to 1. This improves the chances of generating a pattern whose loop can be interchanged. The loop bounds have also decreased since the loop depth is now 3. Otherwise tests may run for too long. Since the loop depth is 3, there are $3! = 6$ possible ways to interchange the loop. There is a high chance that an interesting case will appear when we generate 4 mutations per instance.

When generating each mutation, the transformer ramdomly picks one loop nest permutation from all possible permutations of the loop. If a permutation is invalid, meaning that it is illegal to interchange loops to reach that permutation, the transformer tries again until a valid permutation is found. There will always be at least one valid permutation which is the original permutation. Duplicate mutations are allowed. They can either inflate or deflate the stability results if the majority of mutations are fast or slow, respectively, but we believe that the inflation and deflation would have equal chances of happening and would cancel out for a large number of patterns. Additionally, keeping the structure of the patterns constant for all data points, by enforcing all patterns to have the same number of instances and all instances to have the same number of mutations, increases the reliability of our statistical measurements. However, if a mutation group consists of only duplicates, we discard that mutation group since its stability will be 1 and will only inflate the stability metrics.

No compiler had incorrectness bugs for this study.

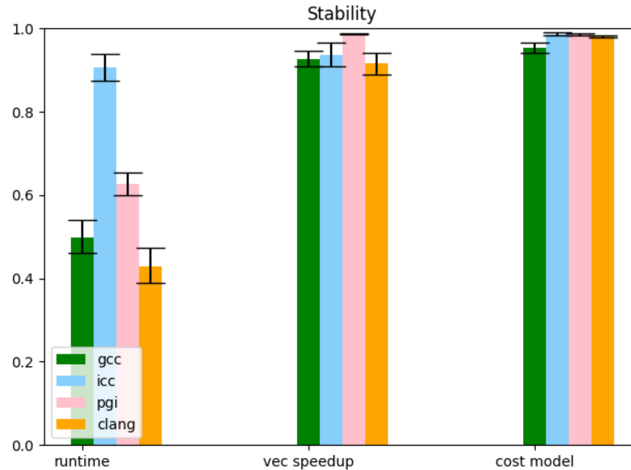


Figure 4.14: Stability results for loop interchange study

The stability results are shown in figure 4.14. For this study, ICC outperformed other compilers in runtime stability. All compilers had high vector speedup and cost model stability..

Figures 4.15, 4.16, and 4.17 show the distribution of runtimes, vector speedups, and cost model speedups, respectively.

While ICC performed the best, extreme outliers still existed. A superficial inspection at the assembly of the slowest and fastest mutations show that the slower mutation used vector instructions heavily and seemed to be unrolling some loops, while the faster mutation used simple scalar instructions with no loop unrolling. The performance gain of interchanging with the correct permutation outperformed vectorization in this case. In many cases, the compiler made drastically different decisions for different loop ordering.

Figures 4.18 and 4.19 show the top/bottom and better rank proportions, respectively. Overall ICC performs better than other compilers. Compilers GCC and Clang have comparable performance, indicated by a large gap between GCC and Clang in figure 4.19. PGI is the least performant in this study.

When looking closer at the speedups in figure 4.20, when ICC performs well, it performs extremely well, resulting in speedups at least 4x and up to 6x. None of the other compilers had enough data points for speedups against ICC. Clang not only had a higher proportion of tests that were faster than GCC, its average speedup is also better. Finally, both GCC and Clang had higher proportions of faster tests than PGI and their speedups were significant, up to x2 and x3 for GCC and Clang, respectively.

The distributions of peer speedups are shown in figures 4.21, 4.22, 4.23, and 4.24.

While PGI overall performed the worst, there are still extreme outliers that perform better

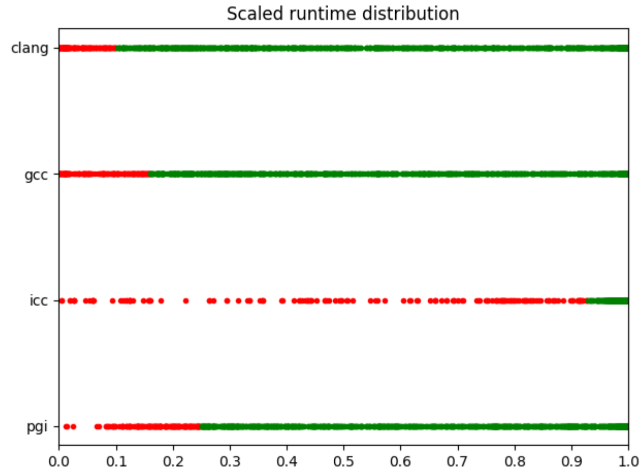


Figure 4.15: Scaled runtime distribution of loop interchange study

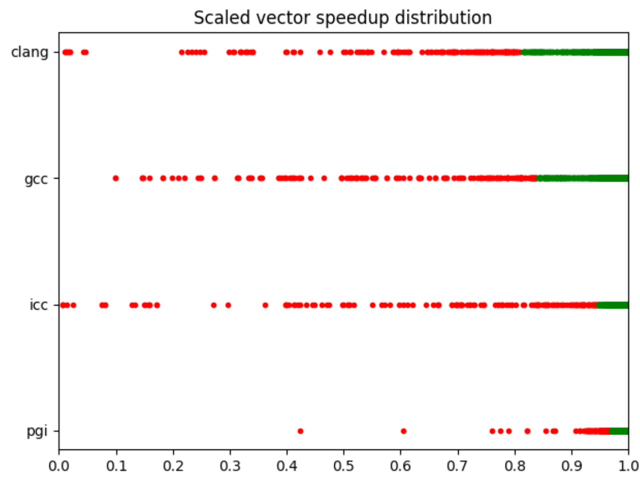


Figure 4.16: Scaled vector speedup distribution of loop interchange study

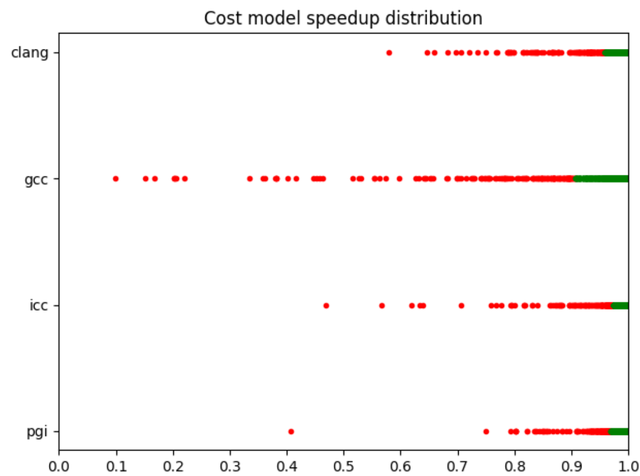


Figure 4.17: Scaled cost model distribution of loop interchange study

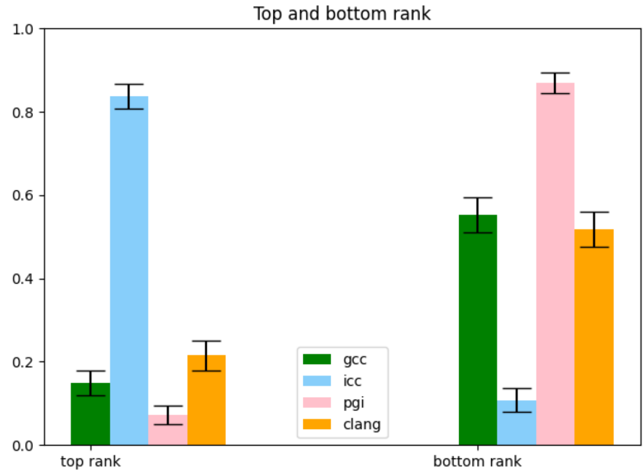


Figure 4.18: Top and bottom rank proportions for loop interchange study

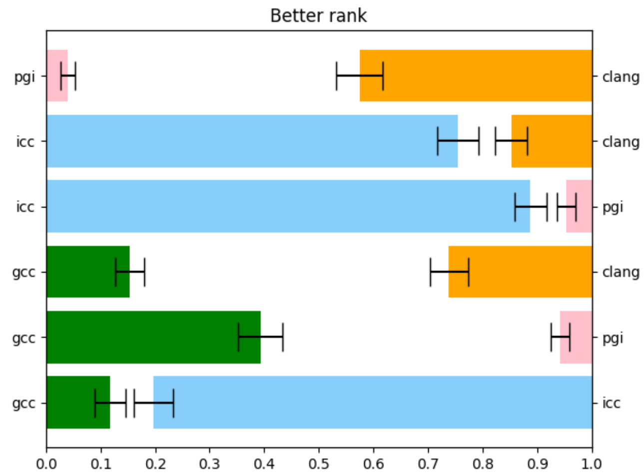


Figure 4.19: Better rank proportions for loop unroll study

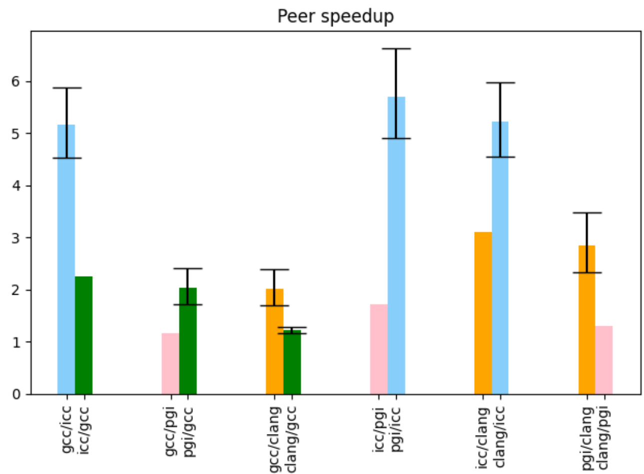


Figure 4.20: Peer speedup for loop interchange study

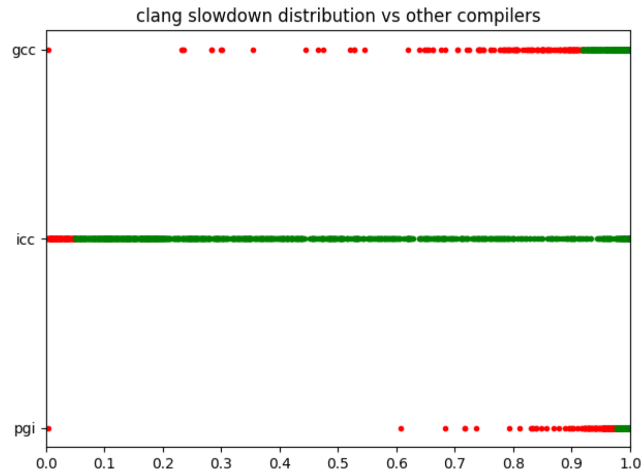


Figure 4.21: Clang's slowdown distribution for interchange study

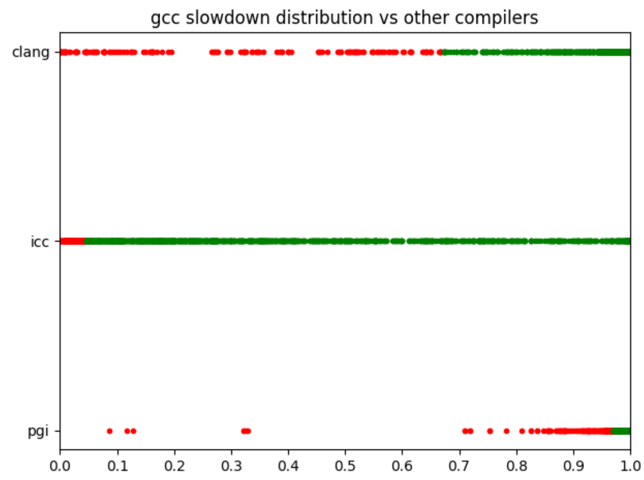


Figure 4.22: GCC's slowdown distribution for interchange study

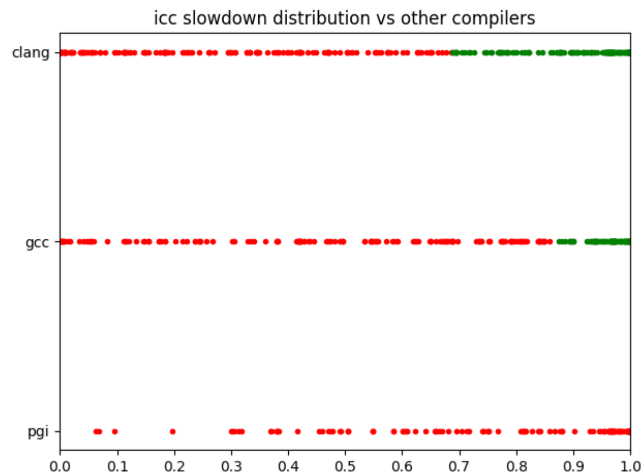


Figure 4.23: ICC's slowdown distribution for interchange study

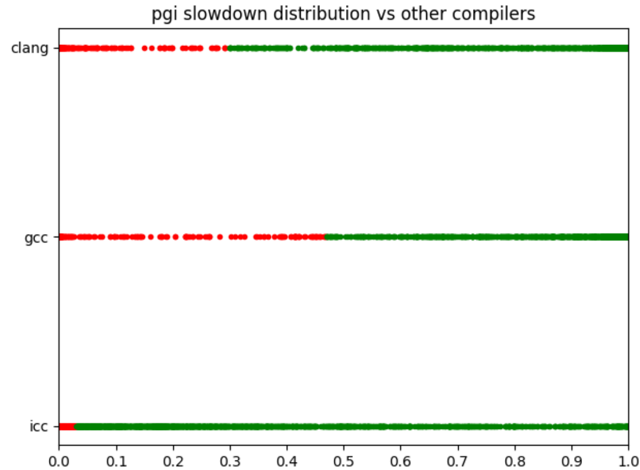


Figure 4.24: PGI's slowdown distribution for interchange study

than ICC. An example is shown in figure 4.25.

```
// only output dependences to the same statements
for (int i2 = 134; i2 <= 292; i2+=1) {
  for (int i3 = 164; i3 <= 234; i3+=1) {
    for (int i1 = 190; i1 <= 225; i1+=1) {
        A[1 * i3 + 1][1 * i3 + 7][2 * i2 - 1] = ...
        A[2 * i3 + 6][1 * i2 + 6][1 * i2 + 7] = ...
    }
  }
}
```

Figure 4.25: Example ourlier where PGI performs much better than ICC

The loop variable i1 is never used in the array subscripts. A quick inspection on the assembly code shows that PGI has two jump instructions, while ICC has three. This shows that, ICC was not able to optimize away loop i1, while PGI did.

4.8 COMPILER EVALUATION: LOOP UNROLL AND JAM

Our third study was loop unroll and jam. Unroll and jam can be viewed as unrolling the outer loop nests, while only spelling out the innermost loop body. We generated 200 patterns, with 2 instance per patterns and 4 mutations per instance. Our pattern and instance profiles are the same as the profiles for the loop interchange study (Tables 4.6a and 4.6b).

The transformation picks one of the (two) outer loops randomly and unrolls it between 1 and 16 times. There are 32 possible mutations per instance. As in the the loop interchange

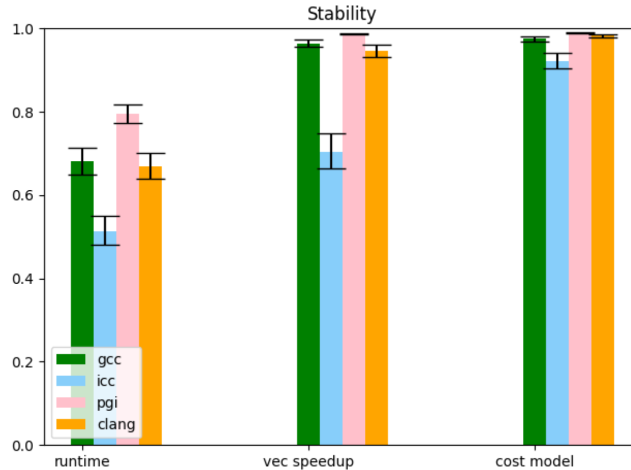


Figure 4.26: Stability results for loop unroll-and-jam study

study, duplicates are allowed as long as not all 4 mutations of the same mutation group are duplicates. However, they are less likely to happen in this study because of the larger number of choices.

One pattern was excluded from the study since ICC encountered a SEGFAULT while compiling one test for all the compilation modes. The other mutations in the same mutation group compiled without any problem. The problematic mutation unrolled-and-jammed the outermost loop 13 times. The other mutations included unrolling-and-jamming the outermost loop 7 and 16 times and the middle loop 12 times.

For this study, PGI was the most stable, GCC and Clang were comparable, and ICC was the least stable, as seen in Figure 4.26.

Figures 4.27, 4.28, and 4.29 all show the presence of extreme outliers, especially for ICC. An extreme outlier for ICC includes a case where the version with an unrolled middle loop (with factor 13) was x150 faster than the version with no unrolling. We looked at the assembly code, and found that the mutation with unrolled-and-jammed mutation had better reuse of registers.

The top/bottom, and better rank proportions are shown in Figures 4.30 and 4.31. ICC had the best performance, Clang performed slightly better than Clang, and PGI performed the worst.

The average speedup and distributions are shown in Figures 4.32, 4.33, 4.34, 4.35, and 4.36. ICC still performs better than all other compilers, but not as well compared to its performance in the loop interchange study. Clang performed quite well against other compilers except ICC.

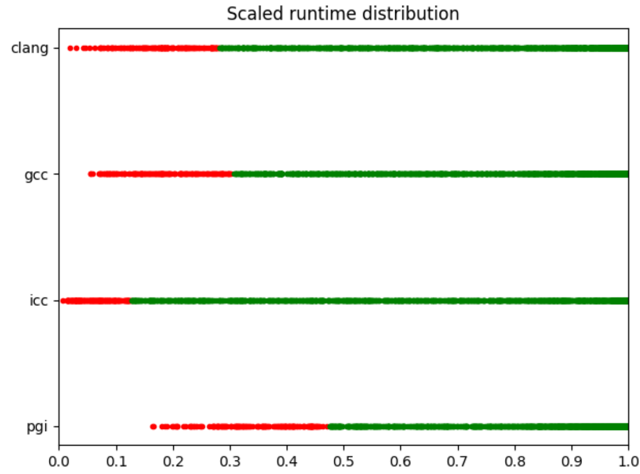


Figure 4.27: Scaled runtime distribution of loop unroll-and-jam study

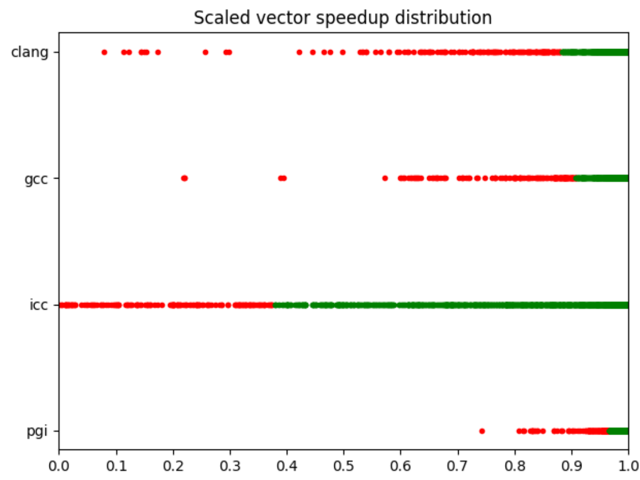


Figure 4.28: Scaled vector speedup distribution of loop unroll-and-jam study

Figure 4.29: Scaled cost model distribution of loop unroll-and-jam study

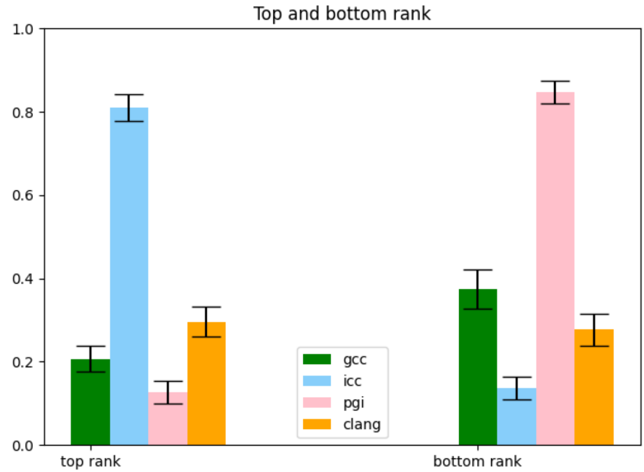


Figure 4.30: Top and bottom rank proportions for loop unroll-and-jam study

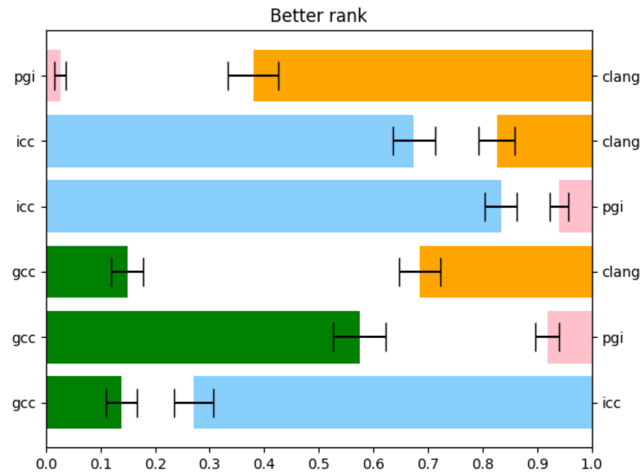


Figure 4.31: Better rank proportions for loop unroll-and-jam study

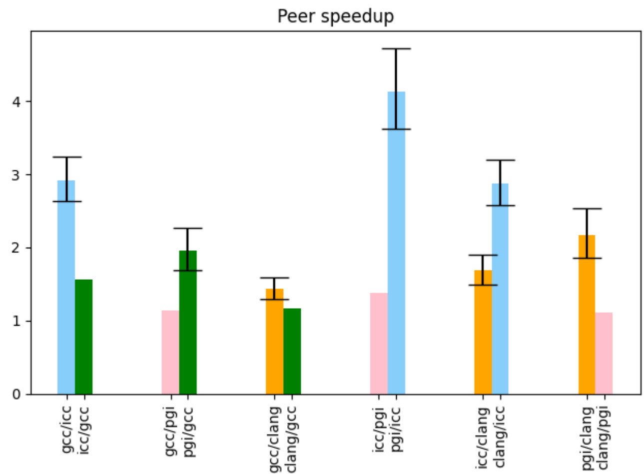


Figure 4.32: Peer speedup for loop unroll-and-jam study

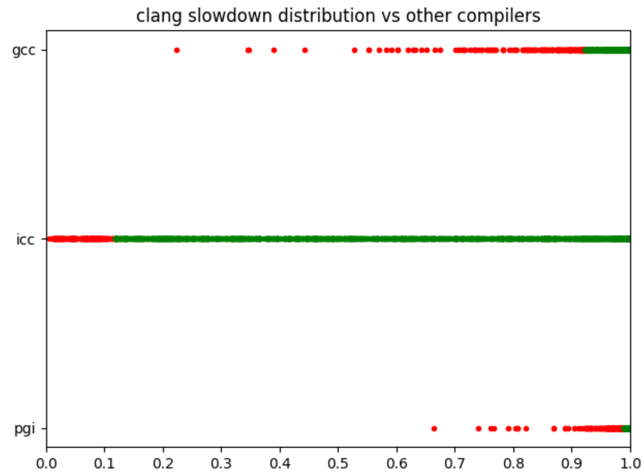


Figure 4.33: Clang's slowdown distribution for loop unroll-and-jam study

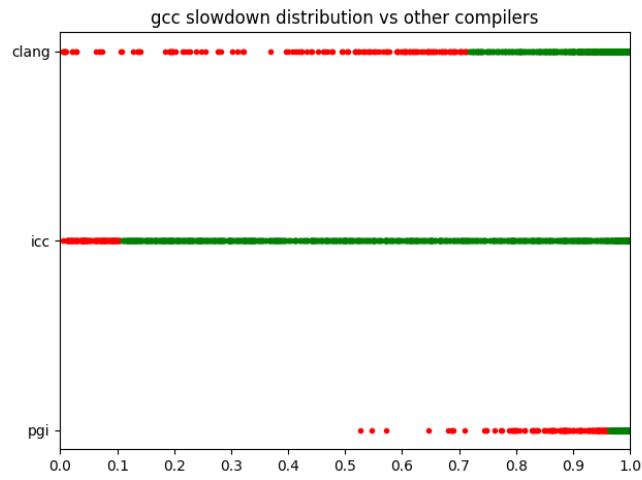


Figure 4.34: GCC's slowdown distribution for loop unroll-and-jam study

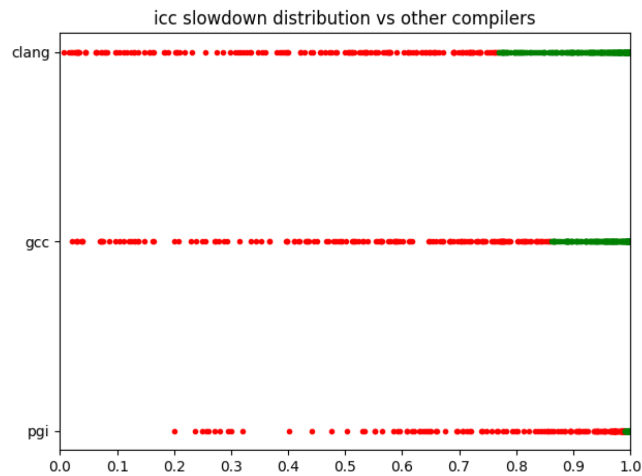


Figure 4.35: ICC's slowdown distribution for loop unroll-and-jam study

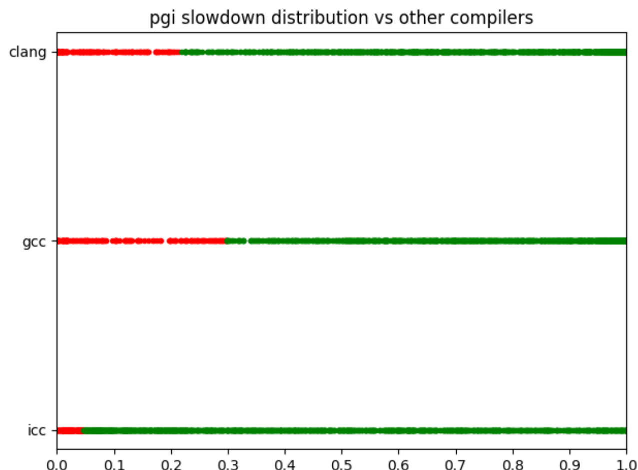


Figure 4.36: PGI's slowdown distribution for unroll-and-jam study

4.9 REPRODUCIBILITY OF EXPERIMENTS

We have emphasized the importance of reproducibility and we went on great lengths to detail the methodology used in order to achieve reliable statistical results.

In this section we show that our experiments are reproducible. We re-ran the unroll, interchange, and unroll-and-jam experiments four more times, totally five runs per experiment, each with newly generated patterns, instances, and mutations.

The random sampling was controlled as we discussed in section 4.3. In particular, the new random tests are generated from the same pattern, instance, and mutation profiles we used in the original experiments. The number of patterns, instances per patterns, and mutations per instances were also controlled to be the same in order to replicate the original data sampling.

For all the studies, all the metrics, and all the compilers, the confidence intervals had narrow ranges, in many cases much less than 0.1, with high overlap. This means it is highly likely that somebody who reproduces our experiments will also get similar narrow intervals and they will likely also overlap, since 95% of them would cover the true attribute of the population.

CHAPTER 5: THE REPRESENTATIVENESS OF SYNTHETIC PROGRAMS

Through random sampling, we are able to estimate properties, such as the mean, of a population through a sample. All of our metrics are mean-based since they are all the mean of some measurement. Therefore, we can use statistical methods to infer the metrics of the a population of programs from a random sample of that population. However, there is no guarantee that the programs defined by the pattern, instance, and mutation profiles represent real-world applications. The evaluation results would be more useful to compiler developers if evaluating compilers on synthetic programs yield similar results as evaluating them on real-world programs.

In this section, we discuss the comparison between synthetic and real-world programs. In particular, we use statistical methods to measure the similarities between the two populations. Note that results may differ depending on the compiler, the metric, and the population of generated programs. For example, we found that, for GCC, programs generated through loop unrolling have the same runtime stability as real-world programs. On the other hand, for Clang, it is not the case for the same experiment. The reader should be aware that any results discussed in this section are only specific to the compiler, metric, and population we describe. We do not claim anything beyond the described context for each experiment.

The experiments conducted in this chapter are less rigorous that those in Chapter 4. However, we believe that they provide a good starting point and equip interested readers with the methodologies for studying the similarities (or differences) between two populations of programs.

5.1 EXPERIMENT SETUP

We need a set of programs representing real-world programs. For this purpose we sample programs from LORE [9]. As described in Chapter 2, LORE is a collection of various benchmarks gathered from multiple domains. The programs from this loop repository represent real-world programs.

With the follow up study on compiler stability [6], the repository also includes mutations of the original loops. Mutations are generated through loop interchange, tiling, unrolling, unroll-and-jam, distribution, and their combinations.

Our experiment focuses on evaluating how well compilers handle loop unroll. Loop unroll is almost always legal, as long as the unroll factor is less than the iteration count. Because of this, no matter what loop we sample from LORE, there will almost always be mutations that

are generated through unrolling. Since the loops in LORE are in C, our test generator which relies on an internal representation could not directly mutate the loops. For this reason, we rely on LORE having these mutations available.

We had to modify the C programs in LORE slightly for our experiments, namely to use `clock_gettime` and `CLOCK_MONOTONIC` to measure runtimes so that it matches the way we measure runtimes in our experiments (Section 4.1.7). LORE's runtime measurement method is based on RDTSC. Since the C programs in LORE follow a simple template and have predictable code structures, we were able to use simple string replacement with the linux command line tool `sed` to carry out this modification.

Another difference between the programs in LORE and ours is in the way data is initialized. In our approach, we initialize data by randomly assigning values between 0 and 1 to all array elements (Section 4.1.4). On the other hand, LORE directly loads the initial content of the variables onto a specific address on the heap. Doing so requires disabling address space layout randomization [73] and setting compiler flags to disable the generation of position-independent code. We set these extra system and compiler options when measuring the runtimes of LORE programs, but not for the synthesized programs.

We use three compilers, namely ICC, GCC, and Clang, in this experiment. PGI had issues compiling the LORE programs, and we did not investigate these issues further. The versions and flags are the same as described in Section 4.1. When compiling loops from LORE, an additional flag was needed to disable position-independent code. The additional compiler flags needed are shown in Table 5.1.

Compiler	Flag
Clang	`-nopie`
ICC	`-no-pie`
GCC	`-nopie`

Table 5.1: Disabling position-independent code for each compiler

Loops sampled from LORE may use C features that are not supported by our pattern language. In particular, they include data structure field accesses, pointer arithmetic, if-else statements, and are not limited by single-precision floating point types.

We extracted a sample of 200 loops from LORE. The unroll factors used for the mutations are limited by what is available in LORE. All loops have at most 4 mutations that are generated from unrolling. Those mutations include the original version of the loop before unrolling and three additional unrolled mutations with factors 2, 4, and 8. We also observed some loops with less than four mutations. The correctness testing for the loops in LORE

follow a similar process as ours. For each program, the checksums of all output variables are calculated. We then check whether equivalent programs have equal checksums or not. After filtering out loops that do not have 4 mutations and loops with incorrect checksums, we had 181 loops remaining.

We tried to match the collection of code selected from LORE, but from the population of generated programs. We generate 200 patterns, 1 instance per pattern, and 4 mutations per instance. The pattern and instance profiles we use are the same as described in Section 4.4. As for the mutation profile, we always include the original loop and generate three additional unrolled versions of the loop with unroll factors randomized uniformly from the interval [2, 16]. This is slightly different from the other experiment in Section 4.4, where we completely randomize the 4 unroll factors. Out of 200 patterns, ICC had an internal error while compiling 1 mutation, so we removed the pattern. Some mutations had overflows while calculating the checksum, and we excluded them just like in Section 4.4. After removing these problematic patterns, we had 187 patterns remaining.

5.2 COMPARING TWO POPULATIONS

We use two approaches, namely statistical methods and visual comparison, when comparing the populations of synthetic programs and real-world programs. Statistical methods are quantitative and can be automated if needed. On the other hand, visual comparison may give the reader a broader picture and more insight into the data. We discuss both approaches below.

5.2.1 Statistical Methods

When comparing two populations, we have several choices. For our experiments, we compared their means, their medians, the cumulative distribution function (CDF), and the probability density function (PDF). We employ statistical methods to test whether the mean, the median, and the CDF of the two populations are the same or not.

The statistical tests were chosen based on simplicity of implementation as long as our data meets the assumptions of those tests. All of the statistical tests being used are available in Python libraries. There may be other more appropriate tests available. We did not explore any of them. However, we believe that this chapter serves as a good starting point for readers who are curious about the methodologies that may be used to compare two populations and hope to answer similar questions as ours.

To compare the mean of two populations, we use Welch's t-test [74]. The requirement for

this test is that the two populations have normal distributions. We check these assumptions using the QQ plot as described in Section 4.3. We implement the test using Python's Scipy library function `scipy.stats.ttest_ind`. Welch's t-test hypothesizes that the mean of the two populations are the same. The test calculates a p-value, which signifies the probability of the two populations being different by random chance. A high p-value indicates that any difference in the mean of the two populations is highly likely from random noise and not the data itself. Therefore a high p-value indicates that the means can be assumed to be the same. Typically, a p-value higher than 0.05 indicates statistical significance.

To avoid the effect that outliers may have on the mean, we also compare the median of two populations. To compare the median of the two populations, we use Mood's median test [75]. The hypothesis of this test is that the median of two populations are identical. This test has no requirements on the distribution of data. This test is available through Python's Scipy library function `scipy.stats.median_test`. We hope for a high p-value which signifies that there is no evidence in the data indicating that the medians of the two populations are different.

Finally, we also compare the two populations as a whole by comparing their CDFs. We use the two-sample Kolmogorov-Smirnov (KS) test [75]. The hypothesis of this test is that the two samples are drawn from the same population. This test has no requirements on the distribution of data. This test is available through Python's Scipy library function `scipy.stats.ks_2samp`. Again, we hope for a high p-value signifying that there is no evidence indicating the two samples are drawn from different populations.

5.2.2 Visual Comparison

While statistical methods help quantify the similarities or differences between two populations, they result in a single number. This may not be useful for practical purposes, especially when the statistical results indicate that the mean, median, or CDF of the two populations are different. Although they may be statistically different, they may still show enough similarity to allow us to conclude that the two experiments do not produce divergent results. For example, even if we find that the runtime stability of generated programs are different from real-world programs, they may only be slightly different, and it would still be practical to use generated programs for the task of providing a high-level evaluation of compilers. To understand the results better, we also visually plot the CDFs and PDFs of the samples in order to see the overall similarities between the two populations.

5.3 RUNTIME STABILITY

We compare the population of generated programs against the population of LORE loops by comparing the mean, the median, and the distribution of their pattern-level runtime stability $r^{\text{stability}}_{\text{pattern}}$ (defined in Section 4.2.3 Equation 4.7).

The p-values of Welch's test, Mood's test, and the KS test are shown in Table 5.2.

Compiler	Welch's test (mean)	Mood's test (median)	KS test (CDF)
Clang	0.0005	0.0003	< 0.0001
ICC	0.68	0.40	0.22
GCC	0.12	0.14	0.08

Table 5.2: The p-values for various statistical tests

For Clang, there is enough evidence in the sample data to suggest that the mean, median, and distribution of the two populations are statistically different. For ICC and GCC, all p-values are above 0.05, which indicates that there is no evidence in the sample data that suggests that the mean, median, or distribution of both populations are different. In other words, for the purpose of measuring the overall runtime stability of ICC and GCC for handling loop unroll, we may potentially use generated loops and get statistically identical results as using real-world loops. We must note that this similarity only applies to the population described above in Section 5.1. Another caveat is that we have not repeated this experiment multiple times to be more confident about these claims.

Visual representations of the distributions are shown in Figures 5.1, 5.2, 5.3, 5.4, 5.5, and 5.6. We observe that the CDFs of both populations have similarities, and the PDFs have large overlapping. For Clang, even though the statistical tests suggest that the two populations are different, when looking at the figures, we believe that there is good similarity.

5.4 PEER SPEEDUP

Similarly, we can use the same methodology to compare the population of generated programs and real-world programs on a different metric. For this experiment, we compare the mean, the median, and the distribution of their pattern-level peer speedup $ps^{\text{all}}_{\text{pattern}}$, defined as follows.

$$ps^{\text{all}}_{\text{instance}}(c_1, c_2, p, i) = geomean(\{ps(c_1, c_2, p, i, m) \mid m \in \mathbf{M}(i)\}) \tag{5.1}$$

$$ps^{\text{all}}_{\text{pattern}}(c_1, c_2, p, i) = geomean(\{ps^{\text{all}}_{\text{instance}}(c_1, c_2, p, i) \mid i \in \mathbf{I}(p)\}) \tag{5.2}$$

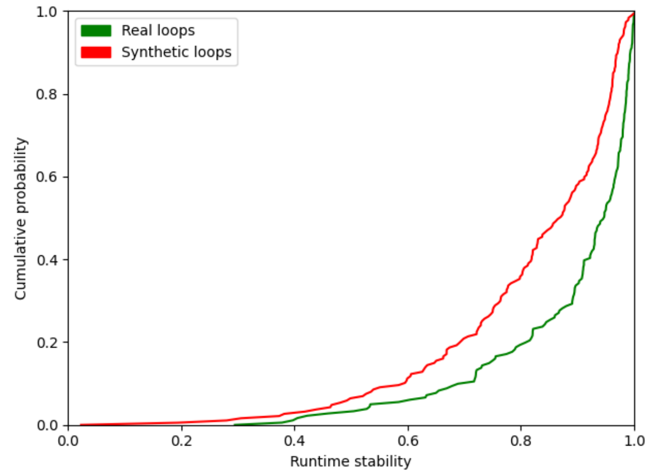


Figure 5.1: CDF of Clang runtime stability

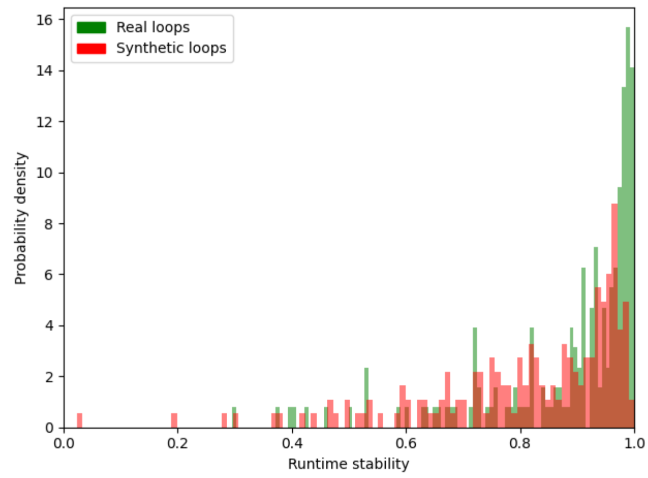


Figure 5.2: PDF of Clang runtime stability

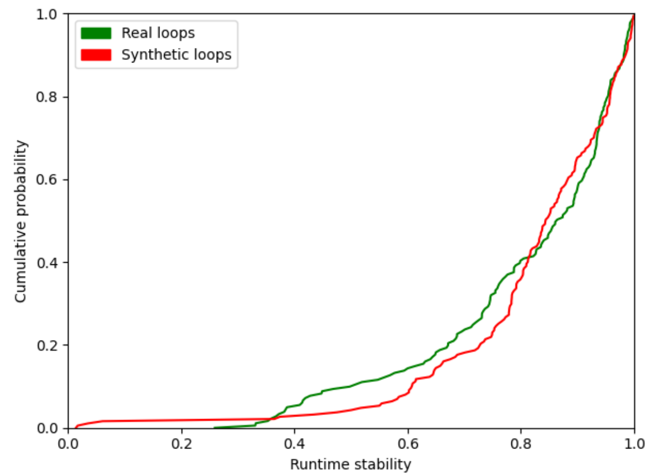


Figure 5.3: CDF of ICC runtime stability

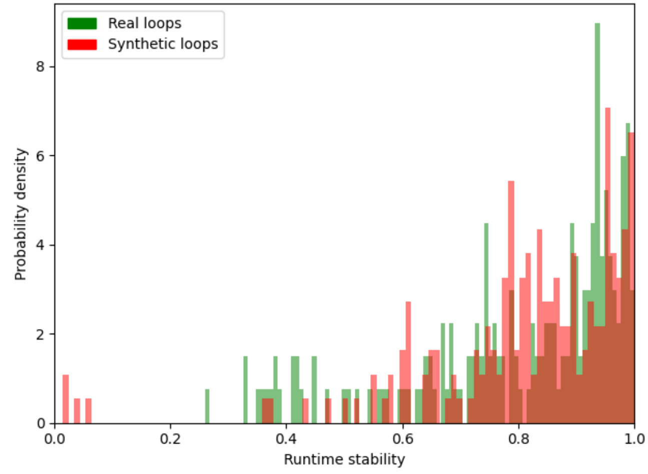


Figure 5.4: PDF of ICC runtime stability

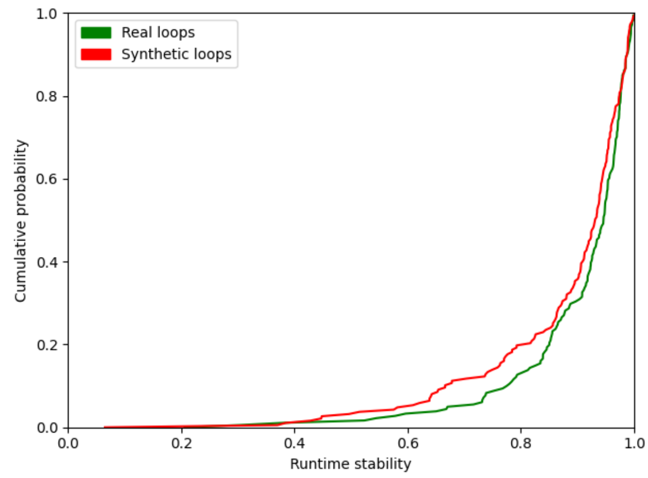


Figure 5.5: CDF of GCC runtime stability

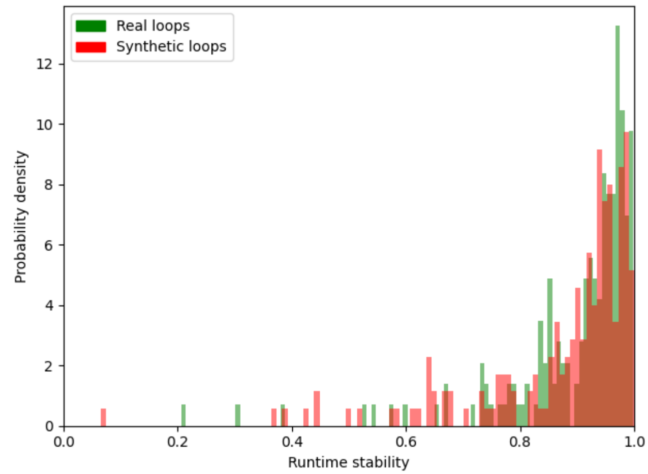


Figure 5.6: PDF of GCC runtime stability

Where ps is the mutation-level peer speedup defined in Section 4.2.10 Equation 4.42.

Note that this is a slightly different definition of the pattern-level peer speedup ps_{pattern} defined in Section 4.2.10 Equation 4.45. In the previous definition, we only included speedups above 1.05x in order to understand the magnitude of speedups when one compiler performs better than another. Here we include all speedups in order to see the overall distribution.

The results of the statistical tests are shown in Table 5.3.

Speedup	Welch's test (mean)	Mood's test (median)	KS test (CDF)
ICC over GCC	0.0009	< 0.0001	< 0.0001
ICC over Clang	0.28	< 0.0001	0.0003
GCC over Clang	0.007	0.09	0.0005

Table 5.3: The p-values for various statistical tests

For this experiment, almost all of the attributes being compared were different for all compiler pairs. From the sample data, only the mean of the speedups of ICC over Clang and the median of the speedups of GCC over Clang were considered not different. However, upon visual inspection of the CDF and PDF plots (Figures 5.7, 5.8, 5.9, 5.10, 5.11, and 5.12), we believe that generated loops and synthetic loops are still similar enough for practical purposes.

The methodologies in this section could be used to compare other populations. This is another area with many open research questions. Although our experiments only scratched the surface of answering deeper questions on the comparison between real and synthetic programs, we believe that the methodologies presented open up many possibilities for further studies.

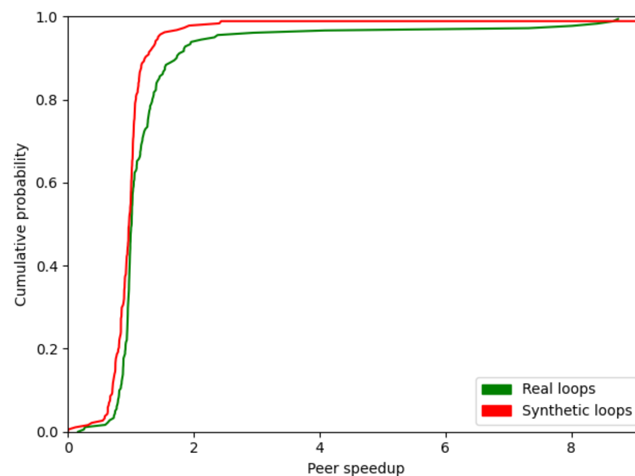


Figure 5.7: CDF of GCC speedup over Clang

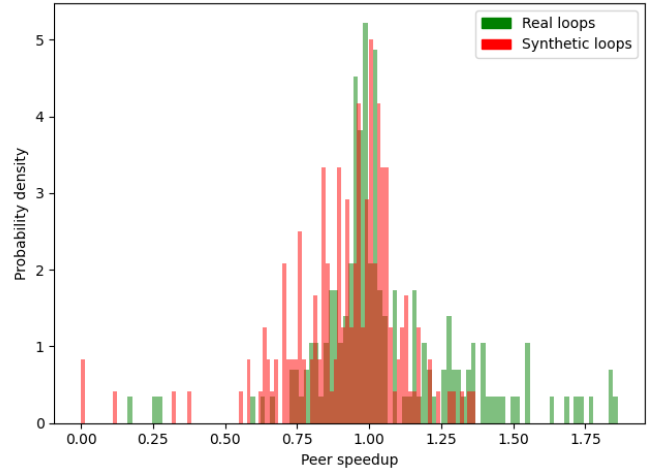


Figure 5.8: PDF of GCC speedup over Clang

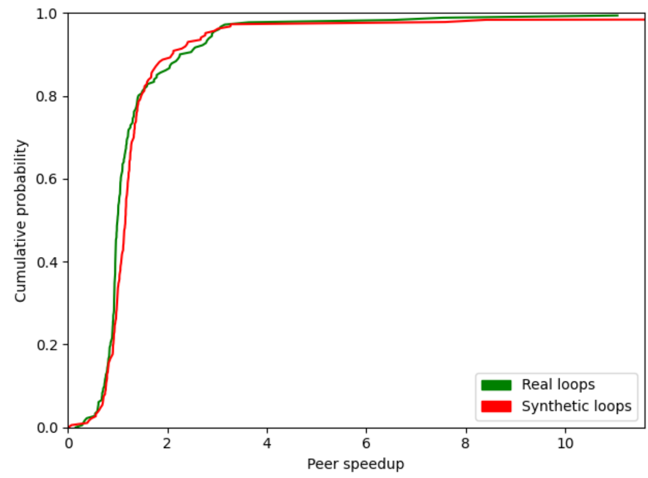


Figure 5.9: CDF of ICC speedup over Clang

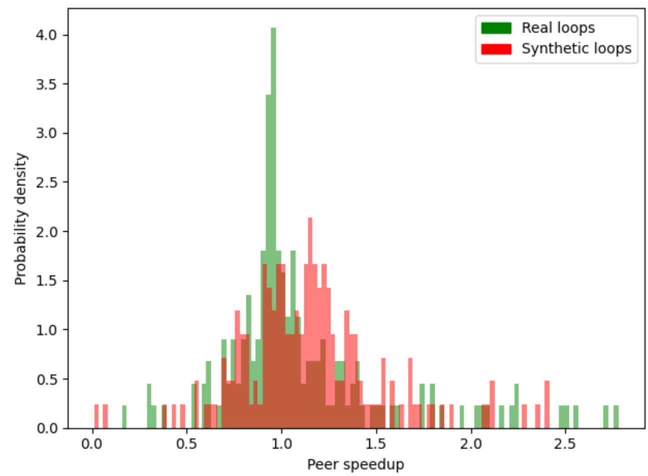


Figure 5.10: PDF of ICC speedup over Clang

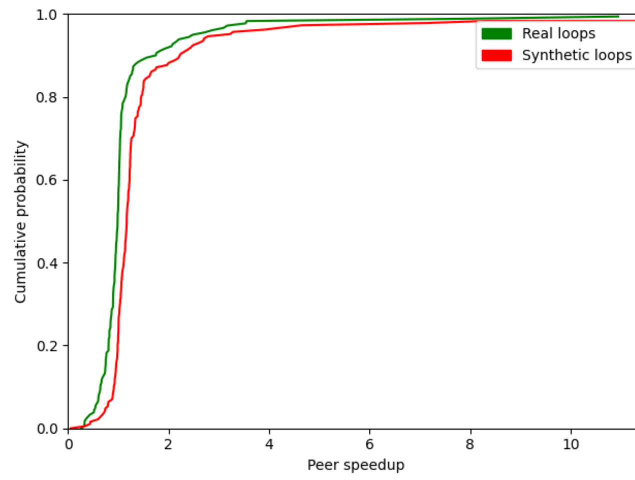


Figure 5.11: CDF of ICC speedup over GCC

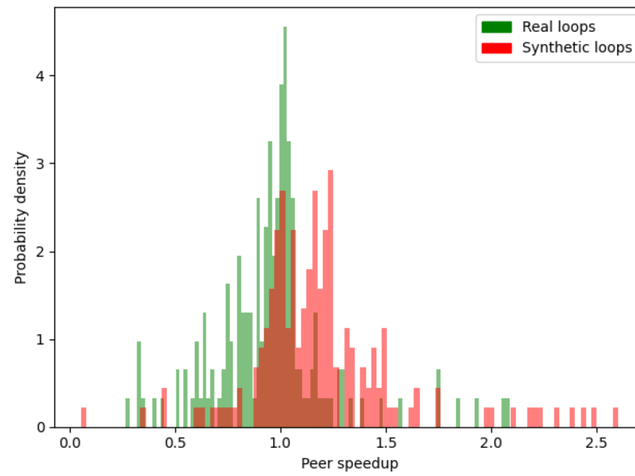


Figure 5.12: PDF of ICC speedup over GCC

CHAPTER 6: TESTING FOR CORRECTNESS

While conducting the experiments in Chapters 4 and 5, we observed a small number of tests that caused compilers to crash. Although the focus of our work is testing for performance, we could potentially use our system to generate tests for correctness. This chapter discusses possible directions to use our system for correctness testing.

In Chapter 2, we discussed related work on correctness testing of compilers in detail. The main differences in each of the approaches are how they generate programs, how they mutate programs, and how they test the compilers. We summarize each approach, including ours, in Table 6.1.

	Approach	Mutation generation	Type of testing
CSmith [2]	Grammar-based	N/A	Differential
Orion [41]	Mutation-based	Delete statements in deadcode regions	Metamorphic
Athena [42]	Mutation-based	Delete & insert statements in deadcode regions	Metamorphic
Hermes [43]	Mutation-based	Insert statements and undo them later in livecode regions	Metamorphic
Our work	Grammar & mutation-based	Use semantic-preserving transformations	Differential & metamorphic

Table 6.1: Different approaches of testing compilers for correctness

In the grammar-based approach, tests are generated based on the grammar (or its subset) of the programming language being used. In the mutation-based approach, tests are generated by mutating existing code. The main distinction in our approach is using semantic-preserving compiler transformations to generate mutations.

In our experiments, we used simple patterns such as singly loop nests with three statements. Our transformations were also simple ones, such as unrolling randomly between 1 and 16 times. Even with simple programs, we ran into two unexpected correctness errors. We believe that with more complicated programs and additional transformations, we will be able to uncover more correctness bugs.

We can make programs more complicated by using more features of the language. Our intermediate representation already supports various operations that were not used in our experiments. We can also add support for other types other than single-precision floating points. Compilers may generate machine code much differently when faced with double-precision floating points, integers, or a mixture of types.

Additionally, we believe that more mutations and their combinations will also lead to programs that trigger more correctness bugs. So far, we have not studied source-level transformations that are not related to loop shapes such as induction variable introduction, scalar expansion, or node splitting. Once we have more transformations, it is also possible to study the effect of transformation chains typically performed in compilers.

While we have not performed any of the aforementioned experiments, the system is already able to support many of them as-is. A more comprehensive experiment needs to be designed and carried out. Detecting correctness bugs may also require a larger number of tests than performance bugs to increase the probability or reaching unexpected corner cases. However, one advantage of testing for correctness over testing for performance is that the correctness tests are not affected by system noise and may be run in parallel on shared memory hardware.

Even though previous studies have reported numerous bugs which have now been fixed, we believe their mutation strategies, such as adding code in dead regions or inserting code that is undone later in live regions, stress different parts of the compiler than our approach. We believe that our work would complement previous studies by exploring bugs related, but not limited, to loop optimization phases and operations on arrays.

CHAPTER 7: CONCLUSIONS

This dissertation sets out to show that random test generation is an effective approach in evaluating compilers when compiling for performance. We presented an infrastructure for carrying out experiments that support this thesis.

There has been a lot of studies on evaluating the correctness of compilers either through formal semantics or testing. However, approaches on evaluating the performance of compilers have not been as well-explored.

One of the main reasons is the vague definition the performance headroom of a program. We define performance to be the ability of compilers to undo source-level transformations that may slow down the execution times of resulting programs. Compilers should compile equivalent programs so that their binaries have the same execution time. If their execution times are different, we know that the compiler can at least produce code that runs as fast as the best-performing program in that set of equivalent programs. Additionally, when different compilers compile the same program and have different execution times, we know that it is possible for that program to run as fast as that compiled by the best-performing compiler on the target machine.

We defined several metrics to quantify performance. These metrics include stability metrics and peer comparison metrics. The stability metrics require sets of equivalent programs to be evaluated using a fixed compiler. The peer comparison metrics require a fixed program to be measured across different compilers.

We generate tests using a 3-stage process. First, we generate patterns which are program skeletons with constant variables. This stage allows us to explore the space of a diverse set of programs. Next, we generate instances from patterns by replacing constant variables with literals. This stage allows us to explore the space of different sizes of the same problem. Finally, we generate mutations which are instances that are transformed through semantic-preserving program transformations typically carried out by compilers. This stage allows us to generate equivalent sets of programs.

We carried out studies that evaluate compilers on how well they handle different types of source-level transformations including loop unroll, loop interchange, and loop unroll-and-jam. PGI and GCC were the most stable when handling loop unroll. ICC was the most stable when handling loop interchange. PGI was the most stable when handling loop unroll-and-jam. ICC performed well in peer comparison for all experiments, seeing average speedups up to x6 over other compilers. For all experiments, all compilers had extreme outliers. For example, even ICC, which performed the best overall, had an ourlier with a

x180000 slowdown after unrolling a loop.

Reproducibility of experiments is of great importance. We described our experiment setup in detail, including, but not limited to, how we initialize random seeds, the compiler versions and flags, how we measure performance, and the population we are sampling from. In addition, we described how to satisfy the data and sampling requirements of, how to use, and how to interpret statistical methods that we use, namely the confidence interval. We repeated our experiments and found all of them to be reproducible.

While randomly generated tests are effective in detecting performance issues in compilers, they might not be useful if the generated tests do not represent real-world programs. We conducted another experiment that aims to quantify how representative are the generated loops. We compared generated loops against loops sampled from the LORE loop repository, which includes benchmarks drawn from real-world programs. We evaluated ICC, GCC, and Clang on how well they handle loop unroll. The metrics we used were runtime stability and peer speedup. We found that for runtime stability, generated loops give similar evaluation results to real-world loops for ICC and GCC, but not Clang. For peer speedup, we found that generated loops did not yield similar evaluation results for any of the compiler pairs. However, when visually inspecting the distribution of these metrics, we believe that the observed differences are acceptable for practical purposes. A more rigorous experiment is required to understand the similarities and differences further. Regardless, we believe the proposed methodology is useful to other researchers.

Finally, we discussed how the test generation infrastructure can also be used to test compilers for correctness. In our experiments, we observe a few miscompilations. We believe that more transformations and the combination of multiple transformations will be able to test compilers more thoroughly and possibly trigger more correctness bugs. Using compiler transformations would be a novel approach to generating mutations compared to previous studies in the literature on testing compiler correctness.

In summary, random test generation is indeed effective in studying and evaluating compilers on the performance of their compiled programs, and we can achieve reproducibility of results by ensuring that the requirements of any statistical methods we use are met.

REFERENCES

[1] X. Leroy et al., "The compcert verified compiler," *Documentation and user's manual. INRIA Paris-Rocquencourt*, vol. 53, 2012.

[2] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *ACM SIGPLAN Notices*, vol. 46, no. 6. ACM, 2011, pp. 283–294.

[3] E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, and Y. Wang, "Intel math kernel library," in *High-Performance Computing on the Intel® Xeon Phi™*. Springer, 2014, pp. 167–188.

[4] G. Baumgartner, A. Auer, D. E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. J. Harrison, S. Hirata, S. Krishnamoorthy et al., "Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 276–292, 2005.

[5] D. A. Padua and M. J. Wolfe, "Advanced compiler optimizations for supercomputers," *Communications of the ACM*, vol. 29, no. 12, pp. 1184–1201, 1986.

[6] Z. Gong, Z. Chen, J. Szaday, D. Wong, Z. Sura, N. Watkinson, S. Maleki, D. Padua, A. Veidenbaum, A. Nicolau et al., "An empirical study of the effect of source-level loop transformations on compiler stability," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, p. 126, 2018.

[7] D. Callahan, J. Dongarra, and D. Levine, "Vectorizing compilers: A test suite and results," in *Supercomputing'88: Proceedings of the 1988 ACM/IEEE Conference on Supercomputing, Vol. I.* IEEE, 1988, pp. 98–105.

[8] S. Maleki, Y. Gao, M. J. Garzar, T. Wong, D. A. Padua et al., "An evaluation of vectorizing compilers," in *2011 International Conference on Parallel Architectures and Compilation Techniques.* IEEE, 2011, pp. 372–382.

[9] Z. Chen, Z. Gong, J. J. Szaday, D. C. Wong, D. Padua, A. Nicolau, A. V. Veidenbaum, N. Watkinson, Z. Sura, S. Maleki et al., "Lore: A loop repository for the evaluation of compilers," in *2017 IEEE International Symposium on Workload Characterization (IISWC).* IEEE, 2017, pp. 219–228.

[10] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Computing Surveys (CSUR)*, vol. 23, no. 1, pp. 5–48, 1991.

[11] R. Hamlet, "Random testing," *Encyclopedia of software Engineering*, 2002.

[12] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *2011 33rd International Conference on Software Engineering (ICSE).* IEEE, 2011, pp. 1–10.

[13] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2123–2138.

[14] L. L. Smith, "Vectorizing c compilers: how good are they?" in *Supercomputing'91: Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*. IEEE, 1991, pp. 544–553.

[15] D. E. Knuth, "An empirical study of fortran programs," *Software: Practice and experience*, vol. 1, no. 2, pp. 105–133, 1971.

[16] M. J. Bassman, G. A. Fisher Jr, and A. Gargaro, "An approach for evaluating the performance efficiency of ada compilers," in *Proceedings of the 1985 annual ACM SIGAda international conference on Ada*, 1985, pp. 151–163.

[17] J. M. Caron and P. A. Darnell, "Bugfind: A tool for debugging optimizing compilers," *ACM SIGPLAN Notices*, vol. 25, no. 1, pp. 17–22, 1990.

[18] D. B. Whalley, "Automatic isolation of compiler errors," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 5, pp. 1648–1659, 1994.

[19] M. Boussaa, O. Barais, B. Baudry, and G. Sunyé, "Notice: A framework for non-functional testing of compilers," in *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2016, pp. 335–346.

[20] A. Pnueli, M. Siegel, and E. Singerman, "Translation validation," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 1998, pp. 151–166.

[21] Q. Carbonneaux, J. Hoffmann, and Z. Shao, "Compositional certified resource bounds," in *ACM SIGPLAN Notices*, vol. 50, no. 6. ACM, 2015, pp. 467–478.

[22] M. Asăvoae, "K semantics for assembly languages: A case study," *Electronic Notes in Theoretical Computer Science*, vol. 304, pp. 111–125, 2014.

[23] S. Dasgupta, D. Park, T. Kasampalis, V. S. Adve, and G. Roşu, "A complete formal semantics of x86-64 user-level instruction set architecture," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 1133–1148.

[24] D. L. Bird and C. U. Munoz, "Automatic generation of random self-checking test cases," *IBM systems journal*, vol. 22, no. 3, pp. 229–245, 1983.

[25] P. Godefroid, M. Y. Levin, D. A. Molnar et al., "Automated whitebox fuzz testing." in *NDSS*, vol. 8, 2008, pp. 151–166.

[26] P. Godefroid, M. Y. Levin, and D. Molnar, "Sage: whitebox fuzzing for security testing," *Queue*, vol. 10, no. 1, pp. 20–27, 2012.

[27] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *2009 IEEE 31st International Conference on Software Engineering.* IEEE, 2009, pp. 474–484.

[28] X. Liu, X. Li, R. Prajapati, and D. Wu, "Deepfuzz: Automatic generation of syntax valid c programs for fuzz testing," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, 2019, pp. 1044–1051.

[29] C. Lidbury, A. Lascu, N. Chong, and A. F. Donaldson, "Many-core compiler fuzzing," *ACM SIGPLAN Notices*, vol. 50, no. 6, pp. 65–76, 2015.

[30] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for c compiler bugs," in *ACM SIGPLAN Notices*, vol. 47, no. 6. ACM, 2012, pp. 335–346.

[31] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.

[32] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide, and J. Regehr, "Taming compiler fuzzers," in *ACM SIGPLAN Notices*, vol. 48, no. 6. ACM, 2013, pp. 197–208.

[33] Q. Zhang, C. Sun, and Z. Su, "Skeletal program enumeration for rigorous compiler testing," in *ACM SIGPLAN Notices*, vol. 52, no. 6. ACM, 2017, pp. 347–361.

[34] B. Daniel, D. Dig, K. Garcia, and D. Marinov, "Automated testing of refactoring engines," in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2007, pp. 185–194.

[35] R. J. Lipton, "Fault diagnosis of computer programs," 1971.

[36] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[37] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE transactions on software engineering*, vol. 37, no. 5, pp. 649–678, 2010.

[38] S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Cortés, "A survey on metamorphic testing," *IEEE Transactions on software engineering*, vol. 42, no. 9, pp. 805–824, 2016.

[39] T. Chen, S. Cheung, and S. Yiu, "Metamorphic testing: a new approach for generating next test cases. technical report hkust-cs98-01," *Hong Kong Univ. of Science and Technology*, 1998.

[40] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.

[41] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," in *ACM SIGPLAN Notices*, vol. 49, no. 6. ACM, 2014, pp. 216–226.

[42] V. Le, C. Sun, and Z. Su, "Finding deep compiler bugs via guided stochastic program mutation," in *ACM SIGPLAN Notices*, vol. 50, no. 10. ACM, 2015, pp. 386–399.

[43] C. Sun, V. Le, and Z. Su, "Finding compiler bugs via live code mutation," in *ACM SIGPLAN Notices*, vol. 51, no. 10. ACM, 2016, pp. 849–863.

[44] A. F. Donaldson, H. Evrard, A. Lascu, and P. Thomson, "Automated testing of graphics shader compilers," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–29, 2017.

[45] M. J. Gardner and D. G. Altman, "Confidence intervals rather than p values: estimation rather than hypothesis testing." *Br Med J (Clin Res Ed)*, vol. 292, no. 6522, pp. 746–750, 1986.

[46] C. Lindig, "Random testing of c calling conventions," in *Proceedings of the sixth international symposium on Automated analysis-driven debugging*, 2005, pp. 3–12.

[47] E. Eide and J. Regehr, "Volatiles are miscompiled, and what to do about it," in *Proceedings of the 8th ACM international conference on Embedded software*, 2008, pp. 255–264.

[48] E. Nagai, H. Awazu, N. Ishiura, and N. Takeda, "Random testing of c compilers targeting arithmetic optimization," in *Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2012)*, 2012, pp. 48–53.

[49] V. Livinskii, D. Babokin, and J. Regehr, "Random testing for c and c++ compilers with yarpgen," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–25, 2020.

[50] T. C. Team, *LibTooling — Clang 12 documentation*, (accessed January 8, 2021). [Online]. Available: https://clang.llvm.org/docs/LibTooling.html

[51] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[52] D. Jovanović, "Solving nonlinear integer arithmetic with mcsat," in *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 2017, pp. 330–346.

[53] S. Verdoolaege, "isl: An integer set library for the polyhedral model," in *International Congress on Mathematical Software*. Springer, 2010, pp. 299–302.

[54] Û. V. Matiâsevič, *Hilbert's tenth problem*. MIT press, 1993.

[55] K. Kennedy and J. R. Allen, *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., 2001.

[56] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton, "Hyper-threading technology architecture and microarchitecture." *Intel Technology Journal*, vol. 6, no. 1, 2002.

[57] J. Charles, P. Jassi, N. S. Ananth, A. Sadat, and A. Fedorova, "Evaluation of the intel® core™ i7 turbo boost feature," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2009, pp. 188–197.

[58] J. H. Wilkinson, "Error analysis of floating-point computation," *Numerische Mathematik*, vol. 2, no. 1, pp. 319–340, 1960.

[59] M. Moscato, L. Titolo, A. Dutle, and C. A. Munoz, "Automatic estimation of verified floating-point round-off errors via static analysis," in *International Conference on Computer Safety, Reliability, and Security*. Springer, 2017, pp. 213–229.

[60] L. Titolo, M. A. Feliú, M. Moscato, and C. A. Muñoz, "An abstract interpretation framework for the round-off error analysis of floating-point programs," in *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 2018, pp. 516–537.

[61] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[62] L. Djoudi, D. Barthou, P. Carribault, C. Lemuet, J.-T. Acquaviva, and W. Jalby, "Exploring application performance: a new tool for a static/dynamic approach," in *Proceedings of the 6th LACSI Symposium*, 2005.

[63] P. J. Fleming and J. J. Wallace, "How not to lie with statistics: the correct way to summarize benchmark results," *Communications of the ACM*, vol. 29, no. 3, pp. 218–221, 1986.

[64] B. Jacob and T. N. Mudge, *Notes on calculating computer performance.* University of Michigan, Computer Science and Engineering Division . . . , 1995.

[65] J. R. Mashey, "War of the benchmark means: time for a truce," *ACM SIGARCH Computer Architecture News*, vol. 32, no. 4, pp. 1–14, 2004.

[66] D. Citron, A. Hurani, and A. Gnadrey, "The harmonic or geometric mean: does it really matter?" *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 18–25, 2006.

[67] J. F. Robison-Cox, "Having a ball with confidence intervals," *Teaching Statistics*, vol. 21, no. 3, pp. 81–83, 1999.

[68] J. M. Bland and D. G. Altman, "Transformations, means, and confidence intervals." *BMJ: British Medical Journal*, vol. 312, no. 7038, p. 1079, 1996.

[69] R. Gnanadesikan and M. B. Wilk, "Probability plotting methods for the analysis of data," *Biometrika*, vol. 55, no. 1, pp. 1–17, 1968.

[70] N. M. Razali, Y. B. Wah et al., "Power comparisons of shapiro-wilk, kolmogorov-smirnov, lilliefors and anderson-darling tests," *Journal of statistical modeling and analytics*, vol. 2, no. 1, pp. 21–33, 2011.

[71] C. A. Field and A. H. Welsh, "Bootstrapping clustered data," *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, vol. 69, no. 3, pp. 369–390, 2007.

[72] T. Koshy, *Catalan numbers with applications.* Oxford University Press, 2008.

[73] P. Team, "Pax address space layout randomization (aslr)," 2003.

[74] M. Delacre, D. Lakens, and C. Leys, "Why psychologists should by default use welch's t-test instead of student's t-test," *International Review of Social Psychology*, vol. 30, no. 1, 2017.

[75] S. Siegel, "Nonparametric statistics for the behavioral sciences." 1956.