Execution Path Tracing as the Basis for Platform-Agnostic Performance Tests

Citation

Permanent link
https://nrs.harvard.edu/URN-3:HUL.INSTREPOS:37377239

Terms of Use
This article was downloaded from Harvard University's DASH repository, and is made available under the terms and conditions applicable to Other Posted Material, as set forth at http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA

Share Your Story
The Harvard community has made this article openly available. Please share how this access benefits you. Submit a story.

Accessibility
Execution Path Tracing as the Basis for Platform-Agnostic Performance Tests

Ian R. Dinwoodie

A Thesis in the Field of Software Engineering
for the Degree of Master of Liberal Arts in Extension Studies

Harvard University
November 2023
Abstract

Over the past several decades, the global reliance on electronics has surged due to ongoing advancements in computational hardware. This pursuit for enhanced performance, driven by its economic impact on productivity, has led to shorter computer lifespans and a rise in electronic waste. It's imperative for software engineers to understand performance characteristics by optimizing software for real-world use, we can potentially prolong computer lifespans and combat the escalating electronic waste issue.

This research introduces Paptools, a novel software performance evaluation tool suite for predicting computational complexity and effectively mapping execution pathways. We designed and developed an instrumenting compiler, an execution path tracing library, an execution path model fitting utility using symbolic regression, and a platform-agnostic performance assertion library.

Paptools accurately predicted performance models for 93% of identified execution paths for a single measurement environment. We proved the platform-agnosticism of performance assertions based on execution pathways with 96% success rate across eight different measurement environments. Furthermore, Paptools provided more
comprehensive insight into the intricacies of run time performance than Google Benchmark, a proven performance analysis utility, underscoring its potential in the field.
Acknowledgements

I offer my sincerest gratitude to my mentor and thesis director, Stratos Idreos, for his guidance and advice. I am grateful for the collaboration and feedback provided by Subarna Chatterjee. Lastly, I would like to thank my research advisor, Hongming Wang, whose insight was instrumental to shaping this project.
Table of Contents

List of Figures ................................................. x
List of Tables .................................................. xi
Chapter I: Introduction .......................................... 1
  1.1 Motivation ................................................. 1
  1.2 Solution .................................................... 2
  1.3 Contributions ............................................. 3
Chapter II: Software Performance .............................. 5
  2.1 Analysis of Algorithms .................................... 5
  2.2 Software Testing ........................................... 7
  2.3 Performance Testing ....................................... 8
Chapter III: Related Work ...................................... 11
  3.1 Traditional Profilers ...................................... 11
  3.2 Performance Annotations ................................ 12
  3.3 Performance Unit Tests .................................. 14
  3.4 Algorithmic Profiling .................................... 15
Chapter IV: Design ................................................................. 17

4.1 Instrumenting Compiler ..................................................... 17
  4.1.1 Abstract Syntax Tree .................................................. 18
  4.1.2 Control Flow .............................................................. 19
  4.1.3 Invoking the Host Compiler ........................................... 23

4.2 Tracing Library ............................................................... 23
  4.2.1 Execution Pathways .................................................... 23
  4.2.2 Data Export ............................................................... 25

4.3 Analysis Library ............................................................. 25
  4.3.1 Path Generalization .................................................... 26
  4.3.2 Binary Expression Trees .............................................. 27
  4.3.3 Symbolic Regression ................................................... 29
  4.3.4 Genetic Algorithm ..................................................... 29

4.4 Assertion Library ........................................................... 31

4.5 Intended Workflow .......................................................... 31

Chapter V: Implementation .................................................... 34

5.1 Instrumentor ................................................................. 34
  5.1.1 LLVM ................................................................. 34
  5.1.2 Clang ................................................................. 35
  5.1.3 CMake ............................................................... 36
List of Figures

1. Abstract syntax tree for the Demo library `IsEven` function 18
2. Graph representation of simple control flow structures 20
3. Graph representation of complex control flow structures 21
4. Graph representation of loop control flow structures 22
5. Loop generalization example 26
6. Example of a binary expression tree 28
7. Overview of the CMake workflow 32
8. Overview of the LLVM compiler infrastructure 34
9. Overview of LLVM-based compiler frontend structure 35
10. Control flow graph for the Demo library `IsEven` function 42
11. Control flow graph for the Demo library `ShiftsToZero` function 43
12. Control flow graph for the Demo library `IsPrime` function 44
13. Control flow graph for the Demo library `Factorial` function 45
14. Control flow graph for the Demo library `QuadraInc` function 46
15. Control flow graph for the Demo library `CubicInc` function 47
16. Graph representation of a recursive Fibonacci number generator 66
List of Tables

1. Primary On-Premises Measurement Environment ........................................ 39
2. Overview of the Demo Library Functions ..................................................... 40
3. Input Known Time Complexities ................................................................. 41
4. Theoretical and Empirical Time Complexities ............................................. 48
5. Multi-Platform On-Premises Measurement Environments ................................ 49
6. Multi-Platform Cloud Measurement Environments ........................................ 49
7. Models for ShiftsToZero Execution Path Two ............................................. 51
8. Models for IsPrime Execution Path Three .................................................. 52
9. Models for QuadraInc Execution Path Two ................................................. 53
10. Profiled Timing with Paptools Disabled ..................................................... 55
11. Profiled Timing with Paptools Enabled ..................................................... 56
12. Comparison of Empirical Time Complexities ............................................. 58
13. Profiled Timing with Google Benchmark Enabled ....................................... 59
Chapter I.

Introduction

1.1. Motivation

Over the last few decades electronics have become a ubiquitous necessity around the globe. The pursuit of advancing computational hardware, such as the biennial doubling of the number of transistors in an integrated circuit as predicted by Moore’s law Moore, 2006, drives a continuous cycle of improved performance. The relationship between performance and productivity provides an economic motivation for such advancements Keyes, 2006. As a result, the average computer lifespan has been steadily decreasing and electronic waste has become one of the fastest growing sources of waste worldwide Babbitt, Kahhat, Williams, and Babbitt, 2009; N. Perkins, Brune Drisse, Nxele, and D. Sly, 2014. Software engineering without a thorough understanding of performance characteristics is complicit in this cycle. By ensuring that software adequately handles real-world scenarios and meet end-user expectations, software engineers may be able to maintain acceptable levels of productivity on older hardware and extend average computer lifespan.

Arora and Barak (2009) contended that ”the efficiency of an algorithm holds
significantly more importance than the technology used to execute it.” According to Stefan, Horky, Bulej, and Tuma (2017), software performance testing has become widely recognized as a vital quality assurance tool for validating system behavior under various conditions, such as load, stress, and endurance testing. While unit testing and performance testing tools have seen increased adoption over the past decade with the availability of popular frameworks like JUnit, evidence indicates that performance testing at the unit level remains relatively scarce Stefan et al., 2017. Although prior research has explored performance unit testing, unit time complexity, and the effects of test doubles, there is a discernible lack of discussion concerning the integration of these concepts. This gap in the literature suggests an opportunity for further research and development.

1.2. Solution

In this thesis we propose a holistic methodology for platform-agnostic performance testing. We also introduce a suite of software tools, Paptools, for use with such methods. Our proposed solution enables software engineers to easily perform the following tasks:

(i) Record execution pathways traversed at run time.

(ii) Fit context-specific performance models to pathway data.

(iii) Specify and evaluate platform-agnostic performance assertions.
Our suite of software tools facilitate these tasks for C/C++ code bases by providing the following components:

(i) *Papinst*: A just-in-time (JIT) instrumenting compiler that injects pathway tracing calls into user binaries.

(ii) *Paptrace*: A tracing library with reflection to record pathway nodes and relevant contextual information.

(iii) *Papan*: An analysis library to produce best-fitting performance models for a given set of tracing data.

(iv) *Papassert*: A library of utilities to simplify the implementation of performance asserts for the produced models.

1.3. Contributions

In this thesis we designed, implemented, and further evaluated the platform-agnosticism of performance assertions based on execution pathways. In addition, we demonstrated that Paptools provides more comprehensive insight into the intricacies of run time performance than Google Benchmark, a state of the art benchmarking tool.

We evaluated both the accuracy and portability of our tools across a variety of hardware and software configurations using a library of functions with varying known complexities. This thesis explored the following research questions:
(i) How closely does the empirical time complexity of a function match the theoretical time complexity?

(ii) How stable is the empirical time complexity across changes in hardware, operating system, and toolchains?

(iii) What are the practical overheads of using this set tools?

(iv) How does this set of tools compare to existing performance analysis tools?
Chapter II.

Software Performance

2.1. Analysis of Algorithms

The analysis of algorithms is a cornerstone topic of computer science that involves examining and comparing the efficiency of different algorithms with respect to their time and space complexity. The primary goal of algorithm analysis is to identify the resources, such as time and memory, required by an algorithm to solve a problem. The undertaking of such analyses is typically performed by developers to make informed choices when selecting or designing algorithms for specific tasks or to optimize the efficiency of existing algorithms in a code base.

To assess an algorithm’s efficiency, its time and space complexity are typically calculated under various scenarios, such as worst-case, average-case, or best-case. Time complexity refers to the measure of how long an algorithm takes to run based on the size or characteristics of its inputs. This relationship between an algorithm and characteristics of its inputs is context dependent; for example, for an algorithm that takes an integer input the relationship could be related to either the value or the bit structuring of the given number. Space complexity is the defined as the amount
of memory space required to solve an instance of the algorithm as a characteristic of its inputs. These complexities provide a foundation for developers to mathematically describe and compare the performance of algorithms.

Big $O$ notation is a widely used mathematical notation for expressing time and space complexity. It describes the limiting behavior of a function as its characteristic arguments tend towards infinity. In other words, Big $O$ notation helps to quantify the asymptotic growth rate of an algorithm’s resource consumption relative to its input characteristics. This standardized representation enables the comparison of different algorithms’ efficiency abstracted from the code that implements it.

The ability to characterize algorithm performance is essential to identify bottlenecks and optimize software. In practice, algorithm analysis can lead to the discovery of more efficient solutions for a given problem or the identification of better-suited algorithms for particular tasks. Additionally, understanding an algorithm’s time and space complexity can help developers make design decisions that balance performance, memory usage, and other factors, ultimately leading to more efficient and scalable software systems.

Moreover, algorithm analysis can also be used to identify the most suitable algorithm for specific hardware or computational environments. For instance, some algorithms may perform better on parallel processing systems or in distributed computing scenarios. By understanding the underlying complexities and characteristics of various algorithms, developers can tailor their solutions to leverage the capabiliti-
ties of the target platform, further enhancing the efficiency and performance of their software.

2.2. Software Testing

Software testing is a vital process in which a software application is evaluated to detect defects and verify its compliance with specified requirements. This evaluation can occur at various stages of the software development life cycle, from the initial development phases to the final release. The primary objectives of software testing include identifying bugs and errors, enhancing the quality and reliability of the software, and ensuring that it meets the needs of its users.

There are several types of software testing, each with its distinct focus and objectives. These types include unit testing, integration testing, system testing, and acceptance testing. Unit testing, for example, concentrates on examining individual components or modules of the software, whereas integration testing emphasizes the testing of interactions between different components. Unit tests are designed to assess the behavior of each unit—a small, self-contained piece of code, such as a function, method, or class—and ensure that it functions as expected in isolation from other parts of the application. System testing evaluates the software as an entire system, while acceptance testing assesses the software from the end-user’s perspective to determine if it meets their needs and expectations.

In late 2001, Smith (2001) introduced the term ”shift-left testing” in an article
for Dr. Dobb’s Journal. Smith explains that “by integrating development and quality assurance earlier and more deeply in your project plan, you can broaden your testing program while reducing manpower and equipment requirements” Smith, 2001. It appears fitting that just over a year later, Beck (2003) published the influential book, *Test-Driven Development: By Example*, promoting the concept of “driving development with automated tests.” Unit tests form the foundation of the Test-Driven Development (TDD) methodology, wherein tests are written prior to implementing the business logic, guiding the design and development of the application based on concrete expectations.

Moreover, unit tests play an essential role in continuous integration and continuous delivery (CI/CD) pipelines, ensuring that code changes undergo thorough testing before being deployed to production.

### 2.3. Performance Testing

Software performance testing involves evaluating the speed, scalability, and stability of a software application under various load conditions. The primary objective of performance testing is to identify bottlenecks and limitations within the software while determining its behavior under different levels of stress and demand. Performance testing is crucial in the software development process, as it helps discover and address performance-related issues before releasing the software into production. Effective performance testing necessitates meticulous planning, appropriate testing
techniques and tools, adequate infrastructure, and collaboration between developers and testers.

Modern performance testing typically comprises load, stress, and endurance testing. Load testing assesses the software under normal and expected loads to evaluate its behavior at routine stress levels. Stress testing exposes the software to extreme loads and conditions to determine limitations, identify bottlenecks, and pinpoint breaking points. Endurance testing subjects the software to continuous loads to evaluate its behavior over time.

Performance tests are usually conducted at the system level, as opposed to the unit level, and are highly sensitive to changes in hardware and load factors such as disk input/output, memory utilization, and CPU utilization/scheduling Vetter and Worley, 2002. To account for this sensitivity, it is common practice to offload performance testing to dedicated machine clusters in production environments Molyneaux, 2009. This approach, however, is generally considered both costly and time-consuming Ding, Chen, and Shang, 2020. To prevent performance testing requests from creating a bottleneck in the development cycle, performance evaluations are often conducted during coarse-grained milestones, such as completed patches, nightly builds, or release candidates Denaro, Polini, and Emmerich, 2004; Molyneaux, 2009.

Consequently, most performance testing utilities do not seamlessly integrate with the highly iterative workflows of modern software development processes, like the red-green-refactor cycle advocated by Test-Driven Development (TDD). Another
notable consequence of conducting tests at the system level is that while they identify failures to meet non-functional requirements, they provide limited insight into the problematic code causing the issue.

As early as 2007 we see shift-left movement begin to encompass performance testing as Woodside, Franks, and Petriu (2007) predicts that ”[u]ltimately, the model and measurement information will be fed back into design tools, so that performance issues are brought to the forefront early in the design process.”
Chapter III.

Related Work

Chapter III of this thesis reviews literature and developments pertinent to this project. By exploring the relevant research, we establish a foundation for our study, identify gaps in the current knowledge, and highlight the unique contributions our thesis aims to make in advancing the understanding of the subject matter.

3.1. Traditional Profilers

The history of formal software performance-analysis tools can be traced back to the 1960s, during the development of IBM’s OS/360 interactive time-sharing environment, Time Sharing Option Lassettre and Scherr, 1972. At that time, the need for performance analysis tools emerged due to the growing complexity of software systems and the realization that software performance had a significant impact on overall system efficiency and user satisfaction.

Software profiling utilities, such as time, prof, and gprof, constitute a small yet renowned subset of tools designed to meet the increasing demand for performance analysis. These tools are primarily used for measuring and collecting data
related to program execution, such as execution time, memory usage, and function call frequency. By providing developers with detailed performance metrics, profiling utilities enable them to identify performance bottlenecks, inefficiencies, and potential areas for optimization.

However, although these tools deliver invaluable diagnostic insights, their usage is predominantly reactive due to their inability to set performance expectations. They are typically employed after the development phase or when performance issues become apparent, which often results in costly and time-consuming changes to the codebase. This reactive approach to performance analysis can lead to suboptimal software designs and missed opportunities for early optimization.

In his book, *User-Centered Requirements Analysis*, Martin (1988) emphasizes the importance of establishing performance targets to effectively determine whether a project’s non-functional requirements are being fulfilled. By setting quantifiable goals for key performance indicators (KPIs), such as response time, throughput, and resource utilization, developers can proactively design software systems with performance in mind, allowing for a more streamlined development process.

### 3.2. Performance Annotations

Performance annotations have become an increasingly important area of focus within the software performance testing community. Rogora, Carzaniga, Diwan, Hauswirth, and Soulé (2020) introduced the Freud utility, a multi-component frame-
work that derives performance models from unit run times and presents them as probabilistic performance annotations. Freud’s approach allows developers to observe the performance of their software in real-world scenarios, accounting for factors such as hardware variability and software dependencies. This level of granularity is invaluable for optimizing code and identifying potential bottlenecks or performance issues.

Building on the concept of performance annotations, Casey and Shah (2020) introduced performance claim annotations (PCAs), which enable programmers to make cost-metric-based assertions about code sections at runtime. This approach allows developers to identify and assess the impact of specific code segments on overall performance, enabling more targeted optimizations and improved software efficiency.

In addition to these developments, research by Bulej et al. (2012) and Horký, Libič, Marek, Steinhauser, and Tůma (2015) explored the use of stochastic performance logic (SPL) for making performance assertions in relative and hardware-independent terms. SPL offers a formal approach to modeling and reasoning about software performance, enabling developers to make informed decisions regarding code optimizations and hardware configurations. By capturing the inherent variability and uncertainty in software performance, SPL-based annotations can provide a more robust and reliable understanding of software behavior across different environments and workloads.

The increasing popularity of performance annotations highlights the impor-
tance of incorporating performance considerations throughout the software development process. By leveraging these techniques, developers can proactively address performance issues, optimize code, and ensure that software systems meet or exceed user expectations in terms of efficiency, responsiveness, and resource utilization. As the field of software performance testing continues to evolve, performance annotations will undoubtedly play a crucial role in helping developers create high-quality, efficient software systems.

3.3. Performance Unit Tests

The existing literature on shifting performance testing earlier in the software development process, or "shift-left," often overlooks the topic of test doubles. Denaro et al. (2004) posited that "performance measurements in the presence of the stubs are good enough approximations of the actual performance of the final application [because] the available components, e.g., middlewares and databases, embed the software that mainly impact performance."

In another noteworthy study, Chatley, Field, and Wei (2019) investigated unit performance testing in virtual time using performance mock objects. The researchers introduced PerfMock, an extension of the Java jMock2 framework, which enabled the embedding of performance simulation models into mock objects. This innovative approach allowed developers to account for the performance cost of a mocked object in virtual time, without waiting for the passage of real time.
However, it is important to note that both of these studies focus primarily on execution time as the performance cost metric. As a result, they do not thoroughly examine the potential impact of test doubles on unit time complexity, leaving room for further exploration and research in this area.

3.4. Algorithmic Profiling

In recent years, an increasing number of researchers have focused on providing performance feedback earlier in the software development cycle. One of the most notable contributions in this domain is the development of the Trend Profiler (trend-prof) utility by Goldsmith, Aiken, and Wilkerson (2007). This utility offers a way to describe the asymptotic behavior of software by estimating empirical computational complexity based on performance trends under realistic workloads. Although trend-prof is a profiler, it is designed to facilitate comparisons between expectations of scalability and empirical observations.

Several other researchers have proposed methods for estimating time complexity models. These approaches include sparse polynomial regression Huang et al., 2010, instrumented statement counts Gulwani, Mehra, and Chilimbi, 2009, symbolic execution Burnim, Juvekar, and Sen, 2009; Luckow, Kersten, and Pasareanu, 2017, classification through gradient boosted trees Sharma, Vohra, Gupta, and Goyal, 2018, and machine learning Sikka et al., 2020, among others.

The growing interest in early-stage performance feedback highlights the impor-
tance of incorporating performance considerations throughout the software development process. By leveraging these techniques, developers can proactively identify and address potential performance issues, optimize code, and ultimately deliver software that meets or exceeds user expectations in terms of efficiency, responsiveness, and resource utilization. As the field continues to advance, it is expected that further innovations will emerge, enabling developers to create more efficient and high-performing software systems.
Chapter IV.

Design

We use this section to describe the underlying theories of the solution and the design of the system. We start with a discussion of the theoretical basis of the solution and end with a detailed description of the design of the system and its components.

4.1. Instrumenting Compiler

In order to collect run time execution data, context capturing calls must be injected into the input source code. Here we focus on the design related to injection process of and leave the design specifics for these context capturing calls to a later section (Section 4.2).

Due to the complex facets of the C++ language — such as macro expansion, parameter pack expansion, and variadic template instantiation — it is not practical to attempt injecting calls into the user code in an unprocessed state. Instead the instrumenting compiler wraps a compiler frontend to interact with a preprocessed version of the user code in the form of an abstract syntax tree (AST).
4.1.1 Abstract Syntax Tree

An AST is a hierarchical representation of a program’s source code structure. This representation is considered abstract because it captures only the functional aspects of the underlying code. ASTs play a crucial role in modern programming languages and compilers, as they provide an intermediate, high-level representation of the source code that simplifies the processes of analysis, modification, and code generation.

Figure 1: Abstract syntax tree for the Demo library IsEven function.

When a compiler or interpreter processes source code, it usually goes through several stages of processing. The first stage is lexical analysis, where the source code is broken down into a sequence of tokens. The second stage, syntactic analysis (parsing), takes these tokens and constructs an AST. The tree-based structure of ASTs allows for efficient traversal and manipulation of the code, making it easier for subsequent
stages of compilation or interpretation. ASTs are commonly used in development tools, such as linters, code formatters, and refactoring utilities. These tools leverage the abstract nature of ASTs to identify patterns, apply transformations, or enforce coding standards without dealing with the syntactic intricacies of the source code.

Each node in an AST represents a language construct, including expressions, statements, or function declarations. The structure of the tree encapsulates the relationships among these constructs, such as the nesting of expressions and statements within a block. Consequently, the AST offers a convenient means of representing code, laying the foundation for generating more specialized structural analyses like control flow analysis.

The instrumenting compiler walks the AST to find injection points and inject calls to the tracing library. In order to collect the necessary trace information, we will want to capture both free and member function calls, builtin operations (e.g., addition operator for primitive types), and control flow nodes.

4.1.2 Control Flow

Control flow refers to the sequence in which statements, instructions, or function calls are executed within a computer program. It directs the program’s path based on conditions and decisions made during run time. The various control structures, including "if-else" statements, loops, and function calls, empower programs to make decisions, repeat actions, and manage the flow of execution.
Figure 2: Graph representation of simple control flow structures. Graphs (a) and (b) represent if and if-else control flow structures, respectively. Graph (c) represents structures with \( n \) decision vertices with an in-degree of one; if-elseif and switch (no pass-through) structures. Graph (d) represents a switch (pass-through) with all possible outward edges rendered. Dotted lines represent a path where

Figure 2 provides a visual representation of the impact of a select few control flow node types.
In computer science, a block denotes a code segment that performs a specific task and is considered a single entity, despite potentially containing numerous statements. Blocks serve to group statements and structurally partition code sections. They can be employed within control structures to outline the actions performed under specific conditions or repetitions. Some programming languages use blocks to create new scopes, meaning objects declared within a block have limited visibility.
Figure 4: Graph representation of loop control flow structures. Graph (a) represents both while and for loops, graph (b) represents do-while loops, and graph (c) represents a generic infinite loop.

Lexical scope is a feature provided by programming languages to define the visibility and accessibility of variables inside a code block. With lexical scope, a variable’s scope is determined by its position within the source code, relative to the block where it is declared. Variables declared in a block are only accessible within that block and any nested blocks, facilitating the creation of context-specific variables with limited accessibility.

In this work, we focus on imperative programming, which employs statements and control structures such as loops and conditionals to outline a step-by-step procedure for the computer to execute. This encompasses widely-used languages like C, C++, Java, and Python.
4.1.3 Invoking the Host Compiler

At the end of instrumentation the instrumenting compiler is responsible for dispatching the requested compile command using the host compiler. Invoking the host compiler for the actual complication process is necessary to produce binaries similar to those produced without the instrumenting compiler. As a result, the only change in the instrumented binary is the additional context capturing calls.

4.2. Tracing Library

A tracing library is necessary to collect and export contextual execution data at run time. Tracing calls capture identifying and descriptive information such as the original source location of the injection, the type of node that is being traversed, and argument values for calls with parameters. The hierarchical nesting of these nodes, based on scope and access order, constructs representations of an execution pathways.

4.2.1 Execution Pathways

Execution pathways are the set of all possible paths that a program can take during execution. These pathways are determined by the program’s control flow structures, which are the mechanisms that control the order in which program statements are executed. Control flow structures can be categorized into two types: choice and loop.

To represent a program with multiple functions \( f_1, f_2, f_3, \ldots, f_n \) and their cor-
responding sets of program statements or basic blocks $S_1, S_2, S_3, \ldots, S_n$, we first define the set $P_{i,j}(x_1, x_2, \ldots, x_k)$ of all possible execution paths through the pair of functions $(f_i, f_j)$ for a given set of input arguments $x_1, x_2, \ldots, x_k$.

Let $P_{i,j}^{(m)}(x_1, x_2, \ldots, x_k)$ represent the $m^{th}$ possible execution path through the pair of functions $(f_i, f_j)$ for the given input arguments. Then, the set $P_{i,j}(x_1, x_2, \ldots, x_k)$ can be represented as:

$$P_{i,j}(x_1, x_2, \ldots, x_k) = \{P_{i,j}^{(1)}(x_1, x_2, \ldots, x_k), P_{i,j}^{(2)}(x_1, x_2, \ldots, x_k), \ldots, P_{i,j}^{(M)}(x_1, x_2, \ldots, x_k)\}$$

Here, $M$ is the total number of possible execution paths through the pair of functions $(f_i, f_j)$ for the given input arguments.

Next, we define the sequence $P_{executed}$ of paths traversed across all pairs of functions and input arguments that are executing for a single invocation of the program. Let $p^{(t)}$ represent the path executed at the $t^{th}$ step in the sequence. Then, $P_{executed}$ can be represented as:

$$P_{executed} = (p^{(1)}, p^{(2)}, \ldots, p^{(T)})$$

Here, $T$ is the total number of steps in the sequence of executed paths.

The expressions for $P_{i,j}(x_1, x_2, \ldots, x_k)$ and $P_{executed}$ together can be used to describe the execution paths of a program with multiple functions and input arguments. Note that these expressions are general representations and will need to be adapted to specific programs and execution scenarios.
4.2.2 Data Export

At exit, the instrumented binary exports the collected data to disk in a format that preserves tree structuring. JavaScript Object Notation (JSON) is a data interchange format that uses human-readable text to store data objects that familiar to software engineers and data scientists.

4.3. Analysis Library

The analysis library should read the JSON trace data and construct a tree structure representing the execution pathways. This library is responsible for performing the loop generalizations between similar paths and generating the best fitting model for the execution pathways. To do this it will need to perform symbolic regression using a genetic algorithm with unary operator, binary operator, and terminal nodes.

As a secondary design decision, the analysis library should expose its interfaces and structures to allow its use in additional tooling.

In the next section we cover the theory of substituting loops with a generalization so that looped paths can be represented by a single path. This is necessary to reduce the number of execution paths that need to be considered when analyzing a program.
4.3.1 Path Generalization

Figure 5: Loop generalization example for the loop formed by cycle \([v_B, v_C, v_D]\). The loop is traversed 1 time in the first execution path and 2 times in the second. The generalization substitutes the loop with a single transfer node \(L_{BD}(x)\), where \(x\) is the number of times the loop is traversed.

Given the control flow graph for a function and execution path traces for said function that vary only in the number of times a loop is traversed, if we walk the control flow graph using the execution path data and record the set edges traversed, we will see that the set of edges is identical. Therefore, the sequence of edges traversed is different, but the set of edges traversed is the same. This means that we can generalize
the execution path by substituting the nodes in the loop with a single transfer node $L(x)$, where $x$ is the number of times the loop is traversed. This is illustrated in Figure 5.

Note that generalization is reversible, meaning we can plug a value of $x$ into the generalized path to obtain the original path. This type of relationship is called a bijection. Therefore if $PG_f : 1, 2 \rightarrow fx$ then $PG'_f : fx \rightarrow 1, 2$.

4.3.2 Binary Expression Trees

Binary expression trees are a data structure that can be used to represent mathematical expressions consisting of unary and/or binary operators. Unary operators are operators which require a single operand such as the negation operator. Binary operators are operators which require two operands such as the addition operator. There tree need does not need to be balanced; each node may have between zero and two children. If infix notation is used to represent the expression then the tree can be traversed in-order to obtain the expression. If prefix notation is used to represent the expression then the tree can be traversed pre-order to obtain the expression. If postfix notation is used to represent the expression then the tree can be traversed post-order to obtain the expression. In this work we will use infix notation to represent the expression. An example of a binary expression tree is provided in Figure 6.

The time complexity of an execution path is the sum of the cost of the nodes traversed. The cost of a node is the number of times the node is traversed. Therefore,
the time complexity of an execution path can be represented as a mathematical expression. For example, given the path $P_{f_1} = \langle v_A, v_B, v_C \rangle$ can be represented as $T_{f_1} = T_A + T_B + T_C$. This means that a time complexity expression tree can be constructed for each execution path using binary expression trees. However, it is not practical to work with path-specific expressions because every iteration length of the loop will result in a different expression. Therefore, we would prefer to work with generalized expressions.

In the previous section we discussed the substitution of loops with a transfer node, but we did not discuss how that transfer node would be defined. In order to define each loop transfer node we must solve for single representative binary expression tree. For this, we turn to symbolic regression.
4.3.3 Symbolic Regression

Symbolic regression is a form of regression analysis that explores various mathematical expressions to identify the most suitable model that aligns with a specific data set. A defining feature of symbolic regression is that initial expressions are formed randomly; no particular model is assumed at the start of the search. While there are several approaches to symbolic regression, we turn our attention to the most common approach: genetic programming.

4.3.4 Genetic Algorithm

A genetic algorithm is a type of evolutionary algorithm that uses a process similar to natural selection to find the best solution to a problem. The process begins with a population of randomly generated solutions. Each solution is evaluated and assigned a fitness score. The solutions with the highest fitness scores are selected to reproduce. The selected solutions are then combined to form new solutions. This process is repeated until a solution with a sufficiently high fitness score is found.

The genetic algorithm requires a genetic representation of the domain of solutions and a fitness function to evaluate the fitness of each solution. In the domain of genetic biology the genetic representation is a chromosome and the fitness function is the ability of the chromosome to survive and reproduce. As is typical for the genetic algorithm in symbolic regression contexts, the genetic representation is a binary expression tree and the fitness function is the ability of the expression tree to fit the
data set such as the root mean squared error (RMSE). Similar to how a chromosome
is comprised of genes, a binary expression tree is comprised of nodes. Each node is
either a terminal node or an operator node. A terminal node is a node that has no
children such as an integer or a variable. An operator node is an operator with either
one (unary) or two (binary) children as previously discussed. To set the stage for
evolution, a starting population must be drawn. For evolutionary programming this
is typically referred to as the initialization process.

For evolutionary algorithms one must also define the reproduction process. In
the genetic algorithm, reproduction is defined by crossover and mutation. Crossover
is the process of combining two solutions to form a new solution. Mutation is the
process of randomly modifying a solution. In the genetic algorithm, crossover is
performed by selecting a random node from each parent and swapping their children.
Mutation is performed by selecting a random node and replacing it with a randomly
generated node.

Lastly, one must also define the selection process. In the genetic algorithm,
selection is defined by tournament selection. Tournament selection is the process
of selecting a random subset of the population and selecting the solution with the
highest fitness score from that subset. Tournament selection is performed twice to
select two parents for crossover.
4.4. Assertion Library

The assertion library provides utilities that allow the user to assert the time complexity of a function. This library should provide ad hoc functionality to an existing test framework without interfering with any existing functionality.

4.5. Intended Workflow

This section details the intended workflow for the proposed solution. While it may be applicable to other workflows, the workflow described in this section was used during the development, testing, and evaluation of the project. Choices about workflow components were made to align with current popular conventions. Specifically, the software was designed for POSIX based machines using CMake as the build system and either GNU or Clang as the host compiler. The implementation of tracing library and its use of C++ functionality necessitates that the host compiler support C++.

The first phase of the workflow begins with instrumenting the users source code. The user modified their build system files to invoke the instrumenting compiler with settings for instrumentation. The build files generated by the build system will have the instrumented executable prefixing compile commands for configured libraries and executables. When the build step of the build system is invoked, the instrumenting compiler is invoked along with the original compile command. The instrumenting compiler instruments the user’s source code alongside the original source, modifies
the compile command to utilize the instrumented file, and invokes the user’s compiler. The instrumented source file may either be retained or discarded depending on the user’s configuration. Instrumented files may be retained for use in debugging or discarded to prevent modifications to the user’s source tree. In addition to the instrumented binaries, the instrumenter will also output a JSON file containing information about the instrumented AST nodes. This phase is illustrated Figure 7.

The second phase of the workflow is generating trace data. The user invokes their test runner which in turn executes the instrumented binaries and generates trace data. The trace data is held in memory until the instrumented binary exits. At exit, the trace data is serialized to disk in JSON format.

The third phase of the workflow is generating representative models from the
collected data. The user invokes the analysis library with both the instrumentation and trace data sets. The analysis library reads the data sets and uses the loaded data to construct and partition tree structure representing the execution pathways. For each path partition, the analysis library uses symbolic regression to find the best fitting model. After the analyses have completed, the analysis library outputs a JSON file containing the best fitting models.

The final phase of the workflow is the evaluation of performance assertions. The user specifies performance assertions in their performance tests using the assertion library. When the user invokes their performance tests the assertion library reads the best fitting models into memory. For each assertion using the assertion library, the expression provided by the user is evaluated using the best fitting models and a boolean is returned. Failed assertions are reported to the user via the test framework they have employed.
Chapter V.

Implementation

5.1. Instrumentor

5.1.1 LLVM

LLVM is a collection of compiler and toolchain tools, designed to allow the creation of frontends for any programming language and backends for any instruction set architecture. Figure 8 illustrates the infrastructure of a LLVM-based compiler which adheres to the popular three-stage compiler structure.

![Figure 8: Overview of the LLVM compiler infrastructure.](image-url)
5.1.2 Clang

Clang (Lattner & Adve, 2004) is a compiler frontend for the C family of programming languages. Figure 9 illustrates the typical pipeline of an LLVM-based compiler frontend. Compiler front ends convert code to an intermediate representation (IR) through parsing, lexing, syntax analysis, and semantic analysis. When used for compilation, Clang produces LLVM IR data which is consumed by the LLVM compiler back end. The compiler back end is responsible for optimization and assembling machine code from the IR. For this project we used a partial workflow with Clang that ends the compilation cycle after the AST has been produced by the lexical analysis step of preprocessing.

![Overview of LLVM-based compiler frontend structure.](image)

The LibTooling library provided by Clang provides interfaces to construct, walk, and manipulate AST abstractions. Additionally, the LibTooling library provides a robust matching and transformation framework for isolating and instrumenting code blocks of interest. Specifically, the instrumenter application imple-
ments a Tool object which invokes a preprocess-only FrontendAction object. The
FrontendAction object creates an ASTConsumer with an instance of our instrument-
ing RecursiveASTVisitor. The RecursiveASTVisitor class is a visitor pattern im-
plementation that allows for selective visitation to registered AST node types. The
registered nodes of interest are function calls, operators, and control flow statements.
When the visitation method for a registered node type is invoked, instrumentation
edits for the node are passed to a singleton Rewriter class instance. The Rewriter
class manages the sequencing of edits so that the final result is a culmination of the
edits requested of it or throws an error if the suggested edits are non-deterministic.

5.1.3 CMake

CMake is a cross-platform suite of tools designed to compile, test, and pack-
age software. CMake is considered a build system generator because it uses simple
platform- and compiler-independent configuration files to generate build files specific
to the users platform and compiler. We provide a simple config.cmake.in file that
gets published in the install directory of the tool. When users import the tool to
CMake using find_package, the functions for instrumenting and tracing are made
available to the user’s CMake project. Figure 7 illustrates CMake’s operational flow.

5.2. Tracing Library

The tracing library defines a Node class to allow for abstract representation of
a tree data structure. The library exposes macros for capturing identifying and con-
textual information about a node. For example, the \texttt{PAPTRACE\_CALLEE\_NODE} macro, which should be placed on the zeroth line of a function, captures the full signature, AST node ID, and argument names and values of the function. At exit, the nodes are serialized to disk as a JSON file.

5.2.1 JSON

The popular open source JSON package \texttt{nlohmann/json} was used by the tracing library to serialize data. Each node is stored as a JSON object which is key-value pair based associative container type typically implemented as a hash table or as an associative array.

5.2.2 Reflection Library

In order to capture contextual information, we needed a library to provide reflection capabilities so that the captured objects could be serialized in a human readable way. The open source GoogleTest framework provides a universal print utility that we use to capture objects as string. The library has built-in support for many common types and containers, including all types and containers from the C++ Standard Library, and can be extended to support custom types.

5.3. Analysis Library

The open-source Anytree Python library provides lightweight and extensible tree data structure abstractions and utilities. We load the traces into tree structures.
Each instrumentation point is a node in a tree. Each tree has a function call node as its root. We use the DEAP library (Fortin, Rainville, Gardner, Parizeau, & Gagné, 2012) for genetic programming. We use SymPy (Meurer et al., 2017), an open-source Python library for symbolic mathematics, to handle binary expression tree handling. The library provides computer algebra system (CAS) features while being written entirely in Python with minimal external dependencies.

NetworkX (Hagberg, Schult, & Swart, 2008) is an open-source Python library for creation, manipulation, and study of graph and network data structures. We use this library to generate a dependency graph for all the loaded nodes which is a directed acyclic graph (DAG) where each node is a function call and each edge is a dependency between two function calls. This dependency graph is used to determine the order in which functions are called so that previously determined relationships can be substituted instead of recalculated.

5.4. Assertion Library

The assertion framework is a collection of Python functions and objects that wrap SymPy objects for ease of use. The assertion functions exposed by the framework invoke Python’s builtin assert function which is supported by the majority of popular Python testing frameworks.
Chapter VI.

Experiments and Results

Experiments must be guided by the questions that they seek to provide answers for. We start this chapter with a formal statement of the research questions that the experiments seek to answer. Subsequent sections detail the methodology and results for each experiment that was undertaken.

6.1. Evaluating Tool Accuracy

The purpose of this experiment was to evaluate the accuracy of the empirical time complexity derived from our best fit models in comparison to known theoretical time complexities.

This initial evaluation of the tool was conducted on-premises with a single machine according to the hardware and software setup detailed in Table 1.

| Table 1: Primary On-Premises Measurement Environment |
|---------------------------------|--------------------------------------------------|
| CPU                | Apple M1 Max APL1105 3.1Ghz                    |
| Cores              | 10 (8 performance, 2 efficiency)              |
| ISA                | ARMv8.5-A                                      |
| Memory             | 32 GB LPDDR5-6400                              |
| Platform           | macOS Ventura 13.4.1                           |
| Toolchain          | Clang 16.0.6                                   |
We implemented a small C++ library, which we refer to as the Demo library, in order to have an input code base with known theoretical complexities. The signature, description, and worst-case time complexity of each function is listed in Table 2.

<table>
<thead>
<tr>
<th>Signature</th>
<th>Time Complexity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool IsEven(int x)</td>
<td>(O(1))</td>
<td>Returns true if the given number is even.</td>
</tr>
<tr>
<td>int ShiftsToZero(int x)</td>
<td>(O(\log x))</td>
<td>Returns the count of right-shift operations performed to reduce the given number to zero or less.</td>
</tr>
<tr>
<td>bool IsPrime(int x)</td>
<td>(O(\sqrt{x}))</td>
<td>Returns true if the given number is prime.</td>
</tr>
<tr>
<td>int Factorial(int x)</td>
<td>(O(x))</td>
<td>Returns -1 for (x &lt; 0) or (x &gt; 31), otherwise returns the factorial for the given number.</td>
</tr>
<tr>
<td>int QuadraticFn(int x)</td>
<td>(O(x^2))</td>
<td>Returns 0 for (x \leq 0), otherwise returns the result of (x^2) increments.</td>
</tr>
<tr>
<td>int CubicFn(int x)</td>
<td>(O(x^3))</td>
<td>Returns 0 for (x \leq 0), otherwise returns the result of (x^3) increments.</td>
</tr>
</tbody>
</table>

In order to generate trace data the instrumented Demo library binary needed to be executed for a range of input values. To emulate standard development practices unit tests were written for each of the functions using the GoogleTest testing framework. Using the CMake integration provided by Paptools, the Demo library was instrumented by Papinst and compiled into a shared library by the host compiler. The unit tests were compiled into a separate executable and linked against the instrumented library. The unit test executable was invoked to allow Paptrace to capture and export the run time trace data.

Papan was executed to generate best fitting path models for the captured trace data. In order to allow the analysis library to reason about uninstrumented calls, such as built-in operations and functions from the standard library, an addition data file of known time complexities was provided as input. The contents of the file
are summarized in Table 3.

<table>
<thead>
<tr>
<th>Function</th>
<th>Mangled Name</th>
<th>Time Complexity</th>
<th>Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>int std::sqrt(int)</td>
<td>_Z4sqrti</td>
<td>$O(1)$</td>
<td>$C_{sqrt}$</td>
</tr>
<tr>
<td>int operator++(int)</td>
<td>B4op++i</td>
<td>$O(1)$</td>
<td>$C_{op++}$</td>
</tr>
<tr>
<td>int operator&gt;&gt;(int, int)</td>
<td>B5op&gt;&gt;=ii</td>
<td>$O(1)$</td>
<td>$C_{op&gt;&gt;=}$</td>
</tr>
<tr>
<td>int operator%(int, int)</td>
<td>B3op%ii</td>
<td>$O(1)$</td>
<td>$C_{op%}$</td>
</tr>
<tr>
<td>int operator*=(int, int)</td>
<td>B4op*=ii</td>
<td>$O(1)$</td>
<td>$C_{op*=}$</td>
</tr>
</tbody>
</table>

6.1.1 Demo Library Control Flow Graphs

In order to facilitate reasoning and discussion about the identified execution paths, we generated CFGs for each of the functions in the Demo library. Joern, an open source code analysis platform based on code property graphs with support for C/C++, was used to generate the CFGs.

The Demo library `IsEven` function consists of a single execution path from entry to exit as illustrated in Figure 10. The theoretical time complexity of this function is $O(1)$ as the input has no impact on the quantity of work performed.
The Demo library `ShiftsToZero` function consists of two execution paths as illustrated in Figure 11. The first path, represented by the annotated edge sequence (1), is traversed for \( x \in \{ n \mid n \in \mathbb{Z} \text{ and } n \leq 0 \} \) where \( x \) is the input value. The theoretical time complexity of this path is \( O(1) \) as the input has no impact on the quantity of work performed.

The second path, represented by the the annotated edge sequence \((1, (2)^{\log x}, 1)\), is traversed for \( x \in \{ n \mid n \in \mathbb{Z} \text{ and } n > 0 \} \). The number of loop iterations performed is determined by \( \log x \) resulting in a theoretical time of \( O(\log x) \).
The Demo library `IsPrime` function consists of five execution paths as illustrated in Figure 12. The first execution path, represented by the annotated edge sequence (2), is traversed for \( x \in \{ n \mid n \in \mathbb{Z} \text{ and } n \leq 1 \} \) where \( x \) is the input value.

The second execution path, represented by the annotated edge sequence (1, 3), is traversed for \( x \in \{ n \mid n \in \mathbb{Z} \text{ and } 1 < n \leq 3 \} \). The fourth execution path, represented by the annotated edge sequence (1, 4, 5), is traversed for \( x \in \{ n \in 2\mathbb{Z} \mid n > 3 \} \). The theoretical time complexity of first, second, and third paths are \( O(1) \) as the input has no impact on the quantity of work performed.

The third execution path, represented by the annotated edge sequence (1, \( 4\sqrt{x} \), 3), is traversed for \( x \in \{ n \mid n \in \mathbb{P} \text{ and } n > 3 \} \). The fifth execution path, represented by the annotated edge sequence (1, \( 4^k \), 5) where \( k \leq \sqrt{x} \) is traversed for \( x \in \{ n \mid n \notin \mathbb{P} \text{ and } n \equiv 1 \mod 2 \} \). These paths may contain up to \( \sqrt{x} \) iterations which corresponds to a theoretical time complexity of \( O(\sqrt{x}) \)
The Demo library **Factorial** function consists of three execution paths as illustrated in Figure 13. The first execution path, represented by the annotated edge sequence (2), is traversed for \( x \in \{ n \mid n \in \mathbb{Z} \text{ and } n < 0 \text{ or } n > 31 \} \) where \( x \) is the input value. The second execution path, represented by the annotated edge sequence (1, 4), is traversed for \( x \in \{ n \mid n \in \mathbb{Z} \text{ and } n = 0 \} \) The theoretical time complexity of first and second paths are \( O(1) \) as the input has no impact on the quantity of work performed.
The third execution path, represented by the annotated edge sequence $(1, 3^x, 4)$, is traversed for $x \in \{ n \mid n \in \mathbb{Z} \text{ and } 0 < n \leq 31 \}$. The path is composed of a single loop that calculates the factorial of the input value and has a theoretical time complexity of $O(x)$.

![Control flow graph for the Demo library Factorial function. Choice control flow edges have been annotated with numbers to facilitate execution path discussion.](image)

The Demo library `QuadraInc` function consists of two execution paths as illustrated in Figure 14. The first execution path, represented by the annotated edge sequence $(1)$, is traversed for $x \in \{ n \mid n \in \mathbb{Z} \text{ and } n \leq 0 \}$ where $x$ is the input value. The theoretical time complexity for this path is $O(1)$ as the input has no impact on the quantity of work performed.
The second execution path, represented by the annotated edge sequence \(((2, 3^x)^x, 1)\), is traversed for \(x \in \{n \mid n \in \mathbb{Z} \text{ and } n > 0\}\). The path is composed of a doubly nested loop that increments a counter by \(x\) with a theoretical time complexity of \(O(x^2)\).

Demo library \textit{CubicInc} function consists of two execution paths as illustrated in Figure 15. The first execution path, represented by the annotated edge sequence (1), is traversed for \(x \in \{n \mid n \in \mathbb{Z} \text{ and } n \leq 0\}\) where \(x\) is the input value. The theoretical time complexity for this path is \(O(1)\) as the input has no impact on the quantity of work performed.

The second execution path, represented by the annotated edge sequence \(((2, (3, 4^x)^x)^x, 1),\)
is traversed for $x \in \{ n \mid n \in \mathbb{Z} \text{ and } n > 0 \}$. The path is composed of a triply nested loop that increments a counter by $x$ with a theoretical time complexity of $O(x^3)$.

![Control flow graph](image.png)

Figure 15: Control flow graph for the Demo library CubicInc function. Choice control flow edges have been annotated with numbers to facilitate execution path discussion.

6.1.2 Performance Assertion Evaluation

In order to test the accuracy of the path models output by the Papan library, platform-agnostic performance tests were written using the utilities from the Papassert library. Each performance test case that was implemented was designed to assert the theoretical time complexity in $Big O$ notation for a single function-path pair. In total the performance test suite contained 15 test cases to span the 15 paths.
identified across the six Demo library functions.

Of the 15 test cases, one test case failed and 14 test cases passed. The failing

Of the 15 test cases, one test case failed and 14 test cases passed. The failing test case was for path five of the IsPrime function. Table 4 summarizes the path-specific empirical time complexities found by Paptools alongside the theoretical time complexities that were determined in the previous section.

Table 4: Theoretical and Empirical Time Complexities

<table>
<thead>
<tr>
<th>Function</th>
<th>Path</th>
<th>Context</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Theoretical</td>
</tr>
<tr>
<td>IsEven</td>
<td>1</td>
<td>( x \in \mathbb{Z} )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>ShiftsToZero</td>
<td>1</td>
<td>( x \in { n \mid n \in \mathbb{Z} \text{ and } n \leq 0 } )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>( x \in { n \mid n \in \mathbb{Z} \text{ and } n &gt; 0 } )</td>
<td>( O(\log x) )</td>
</tr>
<tr>
<td>IsPrime</td>
<td>1</td>
<td>( x \in { n \mid n \in \mathbb{Z} \text{ and } n \leq 1 } )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>( x \in { n \mid n \in \mathbb{Z} \text{ and } 1 &lt; n \leq 3 } )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>( x \in { n \mid n \in \mathbb{P} \text{ and } n &gt; 3 } )</td>
<td>( O(\sqrt{x}) )</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>( x \in { n \mid n \in 2\mathbb{Z} \text{ and } n &gt; 3 } )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>( x \in { n \mid n \notin \mathbb{P} \text{ and } n \equiv 1 \text{ mod } 2 } )</td>
<td>( O(\sqrt{x}) )</td>
</tr>
<tr>
<td>Factorial</td>
<td>1</td>
<td>( x \in { n \mid n \in \mathbb{Z} \text{ and } n &lt; 0 \text{ or } n &gt; 31 } )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>( x \in { n \mid n \in \mathbb{Z} \text{ and } n = 0 } )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>( x \in { n \mid n \in \mathbb{Z} \text{ and } 0 &lt; n \leq 31 } )</td>
<td>( O(x) )</td>
</tr>
<tr>
<td>QuadraInc</td>
<td>1</td>
<td>( x \in { n \mid n \in \mathbb{Z} \text{ and } n \leq 0 } )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>( x \in { n \mid n \in \mathbb{Z} \text{ and } n &gt; 0 } )</td>
<td>( O(x^2) )</td>
</tr>
<tr>
<td>CubicInc</td>
<td>1</td>
<td>( x \in { n \mid n \in \mathbb{Z} \text{ and } n \leq 0 } )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>( x \in { n \mid n \in \mathbb{Z} \text{ and } n &gt; 0 } )</td>
<td>( O(x^3) )</td>
</tr>
</tbody>
</table>
6.2. Evaluating Tool Stability

The purpose of this experiment was to gauge the platform-agnosticism of Pap-tools by evaluating its stability across multiple platforms. The experiment was conducted on four different machines with a total of eight different configurations. In order to account for differences between on-premises and cloud compute instances the experiment was conducted on both types of machines. The experiment measurement environments for on-premises machines are detailed in Table 5. Environment configurations were selected to ensure differences in host compiler toolchain, instruction set architecture ISA, and operating system.

Table 5: Multi-Platform On-Premises Measurement Environments

<table>
<thead>
<tr>
<th>Config. ID</th>
<th>Toolchain</th>
<th>CPU</th>
<th>Cores</th>
<th>ISA</th>
<th>Memory</th>
<th>Platform</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Clang 16.0.6</td>
<td>Apple M1 Max (APL1105) 3.1GHz</td>
<td>10 (8 performance, 2 efficiency)</td>
<td>ARMv8.5-A</td>
<td>32GB LPDDR5-6400</td>
<td>macOS Ventura 13.4.1</td>
</tr>
<tr>
<td>2</td>
<td>GCC 14.0.3</td>
<td>Intel Core i7 (I7-3520M) 2.9GHz</td>
<td>2</td>
<td>x86-64</td>
<td>8GB DDR3L-1600</td>
<td>macOS Catalina 10.15.7</td>
</tr>
<tr>
<td>3</td>
<td>Clang 16.0.0</td>
<td>Intel i7 (I7-3520M) 2.9GHz</td>
<td>2</td>
<td>x86-64</td>
<td>MacOS Catalina 10.15.7</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>GCC 10.4.0</td>
<td>Intel Core i7 (I7-3520M) 2.9GHz</td>
<td>2</td>
<td>x86-64</td>
<td>MacOS Catalina 10.15.7</td>
<td></td>
</tr>
</tbody>
</table>

The experiment measurement environments for cloud machines are detailed in Table 6.

Table 6: Multi-Platform Cloud Measurement Environments

<table>
<thead>
<tr>
<th>Config. ID</th>
<th>Toolchain</th>
<th>CPU</th>
<th>Cores</th>
<th>ISA</th>
<th>Memory</th>
<th>Platform</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Clang 15.0.7</td>
<td>AMD EPYC Milan 2.0GHz</td>
<td>2 (2 CPUs, 1 core each)</td>
<td>x86-64</td>
<td>4GB DIMM</td>
<td>Ubuntu 23.04 Lunar Lobster</td>
</tr>
<tr>
<td>6</td>
<td>GCC 12.2.2</td>
<td>Intel Broadwell</td>
<td>1</td>
<td>x86-64</td>
<td>500MB DIMM</td>
<td>Debian 12.0 Bookworm</td>
</tr>
<tr>
<td>7</td>
<td>Clang 14.0.6</td>
<td>Intel Broadwell</td>
<td>1</td>
<td>x86-64</td>
<td>500MB DIMM</td>
<td>Debian 12.0 Bookworm</td>
</tr>
<tr>
<td>8</td>
<td>GCC 12.2.0</td>
<td>Intel Broadwell</td>
<td>1</td>
<td>x86-64</td>
<td>500MB DIMM</td>
<td>Debian 12.0 Bookworm</td>
</tr>
</tbody>
</table>
In order to derive empirical performance models for each of the measurement environments the steps from Section 6.1 were conducted for each configuration. The resulting empirical models are presented in the sections that follow.

6.2.1 *IsEven* Performance Models

The analysis library produced the following model for the sole execution path of the *IsEven* function across all configurations:

\[ T_{IsEven_1}(x) = C_{IsEven_0} + C_{op} \%
\]

The left-hand side of the equation, \( T_{IsEven_1}(x) \), should be interpreted as the run time cost model for the first path of the *IsEven* function in terms of the input \( x \). The right-hand side contains \( C_{IsEven_0} \) which is a generic constant to account for the calling cost of the function and \( C_{op} \% \) which, as described in Table 3, accounts for the cost of the \texttt{int operator\%\(\text{int}\,\text{int}\)} operation.

The performance test case for this function passed on all configurations.

6.2.2 *ShiftsToZero* Performance Models

The model for the first execution path of the *ShiftsToZero* function was consistent across all configurations and is provided below:

\[ T_{ShiftsToZero_1}(x) = C_{ShiftsToZero_0} \]

Path models for the second path of the *ShiftsToZero* function were inconsistent.
across configurations. The resulting models for $T_{\text{ShiftsToZero}_2}(x)$ are detailed in Table 7.

### Table 7: Models for ShiftsToZero Execution Path Two

<table>
<thead>
<tr>
<th>Config. ID</th>
<th>Path Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$C_{\text{ShiftsToZero}<em>0} + (C</em>{\text{op}&gt;&gt;} + C_{\text{op}++}) \cdot \log(x^{3/2}) + \sqrt{3}$</td>
</tr>
<tr>
<td>2</td>
<td>$C_{\text{ShiftsToZero}<em>0} + (C</em>{\text{op}&gt;&gt;} + C_{\text{op}++}) \cdot (\log(x) + \log(x - 3)/2)$</td>
</tr>
<tr>
<td>3</td>
<td>$C_{\text{ShiftsToZero}<em>0} + (C</em>{\text{op}&gt;&gt;} + C_{\text{op}++}) \cdot \log(x \cdot \sqrt{x + 6})$</td>
</tr>
<tr>
<td>4</td>
<td>$C_{\text{ShiftsToZero}<em>0} + (C</em>{\text{op}&gt;&gt;} + C_{\text{op}++}) \cdot \log(x^{3/2})$</td>
</tr>
<tr>
<td>5</td>
<td>$C_{\text{ShiftsToZero}<em>0} + (C</em>{\text{op}&gt;&gt;} + C_{\text{op}++}) \cdot \log(5x - 7)$</td>
</tr>
<tr>
<td>6</td>
<td>$C_{\text{ShiftsToZero}<em>0} + (C</em>{\text{op}&gt;&gt;} + C_{\text{op}++}) \cdot \log(x \cdot \sqrt{x + 1})$</td>
</tr>
<tr>
<td>7</td>
<td>$C_{\text{ShiftsToZero}<em>0} + (C</em>{\text{op}&gt;&gt;} + C_{\text{op}++}) \cdot \log(3x)$</td>
</tr>
<tr>
<td>8</td>
<td>$C_{\text{ShiftsToZero}<em>0} + (C</em>{\text{op}&gt;&gt;} + C_{\text{op}++}) \cdot \log(3x)$</td>
</tr>
</tbody>
</table>

The performance test cases for both execution paths of this function passed on all configurations.

#### 6.2.3 IsPrime Performance Models

The models for the first and second execution paths for the `IsPrime` function were constant across all configurations and can be represented by the following equation:

$$T_{\text{IsPrime}_{1,2}}(x) = C_{\text{IsPrime}_0}$$

Path models for the third path of the `IsPrime` function were inconsistent across configurations. The resulting models for $T_{\text{IsPrime}_3}(x)$ are detailed in Table 8.
The path models for the fourth path of the \texttt{IsPrime} function were consistent across all configurations and can be represented by the following equation:

\[
T_{\text{IsPrime}_4}(x) = C_{\text{IsPrime}_0} + C_{\text{sqrt}}
\]

The path models for the fifth path of the \texttt{IsPrime} function were consistent across all configurations and can be represented by the following equation:

\[
T_{\text{IsPrime}_5}(x) = C_{\text{IsPrime}_0} + 4C_{\text{sqrt}} + 3C_{\text{op++}}
\]

The performance test cases for the first, second, and fourth execution paths of this function passed on all configurations. The performance test case for the fifth execution path pass on two our of eight configurations. The performance test case for the fifth execution path failed across all configurations.

### 6.2.4 Factorial Performance Models

The first and second execution paths of the \texttt{Factorial} function were consistent across all configurations and can be represented by the following equation:
The path models for the third path of this function were consistent across all configurations and can be represented by the following equation:

\[ T_{\text{Factorial}_3}(x) = C_{\text{Factorial}_0} + (C_{\text{op}++} + C_{\text{op}++}) \cdot x \]

The performance test cases for all execution paths of this function passed on all configurations.

6.2.5 QuadraInc Performance Models

The first execution path for the QuadraInc function was constant across all configurations and can be represented by the following equation:

\[ T_{\text{QuadraInc}_1}(x) = C_{\text{QuadraInc}_0} \]

The path models for the second path of this function were inconsistent across configurations. The derived models for \( T_{\text{QuadraInc}_2}(x) \) are transcribed in Table 9.

<table>
<thead>
<tr>
<th>Config. ID</th>
<th>Path Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( C_{\text{QuadraInc}<em>0} + C</em>{\text{op}++} \cdot (2x^2 - 1) )</td>
</tr>
<tr>
<td>2</td>
<td>( C_{\text{QuadraInc}<em>0} + 2x^2 \cdot C</em>{\text{op}++} + x \cdot C_{\text{op}++} )</td>
</tr>
<tr>
<td>3</td>
<td>( C_{\text{QuadraInc}<em>0} + C</em>{\text{op}++} \cdot (2x^2 - 1) )</td>
</tr>
<tr>
<td>4</td>
<td>( C_{\text{QuadraInc}<em>0} + C</em>{\text{op}++} \cdot (2x^2 - 1) )</td>
</tr>
<tr>
<td>5</td>
<td>( C_{\text{QuadraInc}<em>0} + C</em>{\text{op}++} \cdot (2x^2 - 1) )</td>
</tr>
<tr>
<td>6</td>
<td>( C_{\text{QuadraInc}<em>0} + C</em>{\text{op}++} \cdot (2x^2 - 1) )</td>
</tr>
<tr>
<td>7</td>
<td>( C_{\text{QuadraInc}<em>0} + C</em>{\text{op}++} \cdot (2x^2 - 1) )</td>
</tr>
<tr>
<td>8</td>
<td>( C_{\text{QuadraInc}<em>0} + C</em>{\text{op}++} \cdot (2x^2 - 1) )</td>
</tr>
</tbody>
</table>
The performance test cases for all execution paths of this function passed on all configurations.

6.2.6 CubicInc Performance Models

The models for the first execution path of the CubicInc function were constant across all configurations and can be represented by the following equation:

\[ T_{\text{CubicInc}_1}(x) = C_{\text{CubicInc}_0} \]

The path models for the second path of this function were constant across all configurations and can be represented by the following equation:

\[ T_{\text{CubicInc}_2}(x) = C_{\text{CubicInc}_0} + 2C_{\text{op}++} \cdot x^3 + 4C_{\text{op}++} \cdot x^2 + C_{\text{op}++} \cdot x + 2C_{\text{op}++} \]

The performance test cases for all execution paths of this function passed on all configurations.

6.3. Evaluating Tool Overhead

The purpose of this experiment was to evaluate the overhead of introducing Paptools into a development workflow. This experiment was carried out using the on-premises measurement environment detailed in Table 1.

Projects were built using CMake’s default build settings for C++ with no additional specification for a compiler optimization level. Regardless of Paptools integration, all builds produced both a static library for the Demo functions and
a unit test executable. Measurements of disk usage were measured using `du` and `ls` from the coreutils tool suite for directory and file size, respectively. Profiling was conducted using hyperfine, a command line benchmarking tool that includes descriptive statistics in its output.

6.3.1 Workflow Overhead without Paptools

For a traditional workflow without Paptools, the mean build time of the project was 9.91 seconds (SD = 1.35) with a range of 8.94 to 13.31 seconds. The profiled times for individual stages of the build and execution of the unit tests are detailed in Table 10.

<table>
<thead>
<tr>
<th>Time (sec.)</th>
<th>Desc.</th>
<th>Mean</th>
<th>Std. Dev.</th>
<th>Min.</th>
<th>Max.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Build - demo library</td>
<td>0.21</td>
<td>0.019</td>
<td>0.19</td>
<td>0.26</td>
<td></td>
</tr>
<tr>
<td>Build - unit tests</td>
<td>2.09</td>
<td>0.021</td>
<td>2.06</td>
<td>2.12</td>
<td></td>
</tr>
<tr>
<td>Execute - unit tests</td>
<td>0.0019</td>
<td>0.0001</td>
<td>0.0017</td>
<td>0.0025</td>
<td></td>
</tr>
</tbody>
</table>

The entire build directory for this build occupied 13.70 megabytes (MB) of the hard drive. The Demo static library, `libdemo.a`, occupied 2.32 kilobytes (KB) while the unit test executable, `demo_unit_tests`, occupied 1.32 MB of disk space.

6.3.2 Workflow Overhead with Paptools

For a workflow with Paptools enabled, it took 13.33 seconds (SD = 0.64) with a range of 12.66 to 14.93 seconds. The profiled times for individual stages of the
build, execution of the unit tests, and additional Paptools steps are detailed in Table 11.

<table>
<thead>
<tr>
<th>Desc.</th>
<th>Time (sec.)</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Build - demo library</td>
<td>3.10</td>
<td>0.007</td>
<td>3.09</td>
<td>3.11</td>
</tr>
<tr>
<td>Build - unit tests</td>
<td>4.06</td>
<td>0.006</td>
<td>4.05</td>
<td>4.07</td>
</tr>
<tr>
<td>Build - trace library</td>
<td>2.20</td>
<td>0.064</td>
<td>2.12</td>
<td>2.30</td>
</tr>
<tr>
<td>Execute - unit tests</td>
<td>2.00</td>
<td>0.18</td>
<td>1.91</td>
<td>2.38</td>
</tr>
<tr>
<td>Execute - model fitting</td>
<td>2.53</td>
<td>0.035</td>
<td>2.50</td>
<td>2.58</td>
</tr>
<tr>
<td>Execute - performance tests</td>
<td>0.46</td>
<td>0.003</td>
<td>0.45</td>
<td>0.46</td>
</tr>
</tbody>
</table>

The installation directory for Papinst occupied 93.37 MB of disk space. The Papinst executable, *papinst*, accounted for 73.95 MB of that usage. The Python virtual environment used to install the Paptools python libraries occupied 340.94 MB of disk space. Of that directory, the files for Papan occupied 67.82 KB and the files for Papassert occupied 14.32 KB.

The entire build directory for the Demo library build occupied 49.24 MB of the hard drive. The Demo static library occupied 514.23 KB while the unit test executable occupied 3.14 MB of disk space. The *paptrace* library, which was fetched as part of the build process, occupied 2.26 MB of the build directory. Execution of the unit test executable produced a trace data file, *paptrace.json*, that occupied an additional 9.14 MB.
6.4. Comparison with Google Benchmark

The purpose of this experiment is to compare Paptools with a tool that is actively used by the software development community to locally gauge performance. For this experiment we opted to use Google’s Benchmark library; a popular C++ micro benchmarking framework. Importantly, the Benchmark library provides a means of estimating the time complexity of a function. We implemented benchmarks for the Demo library using Benchmark with its default settings using values from one to 30 in increments of three (i.e., \(x \in \{x \mid x = 1 + 3n, 0 \leq n \leq 9\}\)). Similar to the previous experiment, this experiment was carried out using the on-premises measurement environment detailed in Table 1.

Results for the Demo library benchmarks were captured in a JSON file and transcribed to Table 12 alongside the theoretical time complexity and the path-specific complexities determined by Paptools. Google Benchmark only produces the complexity of the highest-order term in \( \text{Big O} \) notation.
Table 12: Comparison of Empirical Time Complexities

<table>
<thead>
<tr>
<th>Function</th>
<th>Path</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Theoretical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Paptools</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Benchmark</td>
</tr>
<tr>
<td>IsEven</td>
<td>1</td>
<td>$O(1)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$O(1)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$O(1)$</td>
</tr>
<tr>
<td>ShiftsToZero</td>
<td>1</td>
<td>$O(1)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$O(1)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$O(1)$</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>$O(\log x)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$O(\log x)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$O(1)$</td>
</tr>
<tr>
<td>IsPrime</td>
<td>1</td>
<td>$O(1)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$O(1)$</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>$O(1)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$O(1)$</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>$O(\sqrt{x})$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$O(\sqrt{x})$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Factorial</td>
<td>1</td>
<td>$O(1)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$O(1)$</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>$O(1)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$O(1)$</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>$O(x)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$O(x)$</td>
</tr>
<tr>
<td>QuadraInc</td>
<td>1</td>
<td>$O(1)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$O(1)$</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>$O(x^2)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$O(x^2)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$O(x^2)$</td>
</tr>
<tr>
<td>CubicInc</td>
<td>1</td>
<td>$O(1)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$O(1)$</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>$O(x^3)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$O(x^3)$</td>
</tr>
</tbody>
</table>

For a workflow with Google Benchmark enabled, it took 16.91 seconds (SD = 0.47) with a range of 16.30 to 17.84 seconds. The profiled times for individual stages of the build, execution of the unit tests, and additional Paptools steps are detailed in Table 13.
Table 13: Profiled Timing with Google Benchmark Enabled

<table>
<thead>
<tr>
<th>Desc.</th>
<th>Time (sec.)</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>Std. Dev.</td>
<td>Min.</td>
<td>Max.</td>
</tr>
<tr>
<td>Build - demo library</td>
<td>0.21</td>
<td>0.008</td>
<td>0.20</td>
<td>0.23</td>
</tr>
<tr>
<td>Build - unit tests</td>
<td>2.16</td>
<td>0.046</td>
<td>2.11</td>
<td>2.28</td>
</tr>
<tr>
<td>Build - benchmarks</td>
<td>6.85</td>
<td>0.059</td>
<td>6.67</td>
<td>6.95</td>
</tr>
<tr>
<td>Execute - unit tests</td>
<td>0.002</td>
<td>0.0002</td>
<td>0.002</td>
<td>0.003</td>
</tr>
<tr>
<td>Execute - benchmarks</td>
<td>56.62</td>
<td>0.17</td>
<td>56.16</td>
<td>56.71</td>
</tr>
</tbody>
</table>

The entire build directory for this build occupied $28.74$ megabytes (MB) of the hard drive. The Demo static library occupied $2.32$ KB, the unit test executable occupied $1.26$ MB, and the benchmark executable, `demo_benchmarks`, occupied $1.15$ MB.
Chapter VII.

Discussion

7.1. Accuracy

In the first experiment we found that the Paptools software was capable of accurately estimating the time complexity of functions in a majority of tested scenarios. Using traces from the demo library unit tests, the tool was able to provide estimates of computational complexity on the same order as the known theoretical complexity of the implementations. Paptools accurately predicted models for 14 out of 15 (93%) execution pathways identified for the Demo library.

The single failing model proved the validity of the assertion utilities implemented by Papassert. The results from this experiment also illustrated that the accuracy of Papan produced models is directly related to the quality of the input. Namely, only two contexts were captured for the failing function-path pair; the fifth path of the \texttt{IsPrime} function. It is likely that the two input values that resulted in this path, 25 and 55, were not sufficiently spread apart for the fitting logic to discern the underlying behavior from just two points. This highlights that Paptools shares a weakness with many other performance tools: the quantity and spread of the input
data determines the accuracy of the output models.

The results from first also show that the Paptools software is capable of identifying all executed pathways through a function. Intuitively, given the methodology underpinning the tool, execution pathways that are not covered by the unit tests are not detected by the tool.

The use of an input file of known complexities allow for a full model to be developed despite the possibility that not every call along the path is instrumented. The ability to substitute known or mock complexities for uninstrumented calls allows this tool to integrate seamlessly with TDD workflows.

7.2. Stability

Our second experiment proved the performance assertions performed by Papassert are truly platform agnostic. The vast majority of test cases (96%) passed over a spread of eight different measurement environments. These measurement environments consisted of both Clang and GCC compiler toolchains of various version, multiple macOS and Linux operating systems, and both Intel and ARM instruction set architectures.

It is worth noting that the genetic algorithm library employed does not currently provide a mechanism to set a threshold for an acceptable model. As a result, the algorithm will run through all generations regardless of if an acceptable fit has been found. This can lead to overfitting; a complex model with marginally better fit
than a simple model found early in evolution may be selected as the best fit. All set-
tings for the genetic algorithm employed were optimized to favor a simplified model
and resist continued expansion, but those options were still unable to prevent the
model from evolving. It is possible that the incorrect models derived for \texttt{IsPrime}
which failed on two of the eight configurations were a result of this overfitting.

The pathways modelled as constant or simple linear functions were identical
across all configurations. The stability of these results can be attributed to optimiza-
tions that were added to the model fitting logic. Specifically, the model fitting logic
has optimizations for constant models and perfect linear models. Before attempting
symbolic regression the fitting logic looks for either an identical set of paths or a
set of paths with a perfect linear relationship. The simple linear relationship was
determined through simple linear regression. If either of these cases are satisfied, the
model is assigned a constant or perfect linear coefficient based model accordingly.
Models that were more complex (e.g., logarithmic) had nearly identical traces but
non-identical performance models. The stability or instability of the resulting models
is a direct result of the analysis employed to derive the model. As we see from the
results, simple linear regression was stable while symbolic regression was not. While
the symbolic regression was unstable, it consistently produced results that were on
the same order of magnitude as the theoretical complexity for a majority of execu-
tion paths. It is worth noting that all pathways with a single trace were treated as
constant as there was no basis to assume otherwise.
As mentioned in the Design section of this thesis, this software was designed for POSIX based machines using CMake as the build system and either GNU or Clang as the host compiler. The demonstration of the platform-agnosticisms of this work was limited to POSIX based machines, but should theoretically extend to machines that are not POSIX compliant.

7.3. Tool Overhead

Experiment three illustrated that there is a cost in both time and space for employing Paptools as a performance testing solution. However, this would be expected as the tool requires additional compilation steps, data files, and analyses.

Integration of Paptools increased the total build time of the project by an average of 3.42 seconds, a 34.51% increase. On average, compilation times for the Demo library and unit test executable took an additional 2.89 and 1.97 seconds, respectively. This corresponds to a 1,376% increase in compilation time for the Demo library and a 94.26% increase for the unit test executable. Compilation for the Paptrace library added an average of 2.20 seconds to the build time.

The mean execution time of the unit test executable took an additional 1.99 seconds, corresponding to a 105,000% increase. The execution steps for the Papan and Papassert tools resulted in an additional 2.99 seconds on average. Therefore, integration of Paptools increased the total time of the execution stage for the development workflow by an average of 4.98 seconds.
In terms of additional disk utilization, integration of Papertools increased the size of the build directory by 35.54 MB, a 259.42% increase. The Demo library and unit test executable sizes increased by 511.91 KB and 1.82 MB, respectively. This corresponds to a 22,076% increase in size for the Demo library and a increase of 137.88% for the unit test executable. Including the installation directory of Papinst and the Python virtual environment containing Papan, Papassert, and their dependencies, the total space occupied by the project increased by 472.18 MB.

Intuitively, not all of the overheads will grow with the project size. For example, fetching the Paptrace library is a static cost that has a higher impact on projects with short build times. In some cases the cost may be additive, such as the overhead of calling the instrumented compiler which is realized for each invocation of the host compiler.

7.4. Compared to Google Benchmark

Integration of Google Benchmark increased the total build time of the project by an average of 7.00 seconds, a 70.64% increase. Compilation for the benchmark executable added an average of 6.85 seconds to the build time. The execution steps for the benchmarks resulted in an additional 56.62 seconds on average. In terms of additional disk utilization, integration of Google Benchmark increased the size of the build directory by 15.04 MB, a 109.85% increase.

In comparison, integration of Google Benchmark incurred significantly less disk
overhead but significantly more execution overhead in comparison to integrations with Paptools. Given the general availability of computer storage and the constant drive for heightened productivity, we believe time consumption is more likely a stronger determining factor in tool selection than hard drive consumption.

From the comparison of empirical cost models in Table 13, it is clearly that a solution employing Google Benchmark to categorize its performance is unlikely to identify performance degradation to paths other than the worst case path. It would be possible to manually determine execution paths and write Benchmarks for them, but this would be a laborious undertaking without any utilities to assist in the identification of execution paths. Comparatively, Paptools was able to identify all execution paths for the Demo library and partition the inputs the resulted in the paths. In addition, Google Benchmark failed to correctly identify the worst case complexity for three out of the six (50%) of Demo library functions for the given set of inputs.

7.5. Future Areas of Research

7.5.1 Support for Recursion

The current is able to identify recursive relationships, but fails to accurately determine a model to fit recursive pattern. A graph of call tree for a recursive implementation of a Fibonacci generator is provided in Figure 16. The SymPy library has some limited support for solving for recursive relationships that adhere to prescribed
7.5.2 Scalability Support

The initial implementation of this software focused solely on providing quality results for individual compile commands. Having demonstrated the effectiveness of these results serially, the product could be enhanced by implementing support for parallel invocations of the instrumenting compiler.
Our series of experiments shed light on the capabilities, performance, and limitations of the Paptools software. In the realm of accuracy, the software has showcased its prowess in estimating the time complexity of functions aligning closely with known theoretical complexities with success rates of 93%. The tool’s unique distinction of pathways based on loop exit positions and its ability to pinpoint all executed pathways within functions further underscores its precision. While the paptools software displayed consistent stability across multiple configurations, there were nuances in how different models were derived, especially when delving into the realms of symbolic regression. Regardless, the demonstrated 96% success rate across eight different measurement environments underscores the tool’s potential in the field.

Intuitively, integration of the Paptools software incurs overheads for both disk space and build time. Some of these overheads are static, such as fetching dependencies, while others are incurred on a per-file basis, such as intercepting calls to the host compiler.

When compared with Google Benchmark, another performance evaluation
tool, Paptools offers a more comprehensive insight into the intricacies of a target software’s performance. Integration with Paptools consumed more disk space than Google Benchmark, but provided performance analysis results in a fraction of the time. Furthermore, Paptools provides utilities for the user to make performance assertions the results, a feature that is absent from Google Benchmark.

The methods we presented in this thesis pave the way for execution path based performance analysis. The results presented by this novel solution prove the feasibility of platform-agnostic performance metrics. We believe that integration of these tools can provide useful insights in software development for devices regardless of platform. Performance mindfulness during development may provide an avenue for increasing the lifespan of electronics and, in turn, contribute to a reduction in global electronic waste.
References


Appendix A.

Code Repositories

The code for this project is collected under the Paptools organization on GitHub. Each tool of the software suite was assigned its own repository. These repositories are listed below:

Papinst Repository: github.com/paptools/papinst

Paptrace Repository: github.com/paptools/paptrace

Papan Repository: github.com/paptools/papan

Papassert Repository: github.com/paptools/papan