THE UNIVERSITY *of* EDINBURGH

Inlining in the Glasgow Haskell Compiler: Empirical Investigation and Improvement

Celeste Hollenbeck

Doctor of Philosophy

THE UNIVERSITY OF EDINBURGH

2025

Abstract

Inlining has been claimed to be the single most important optimization in Haskell, and also the optimization receiving the most attention. It is also considered one of the hardest optimizations to get right. If done poorly, it may cause code bloat, cache misses, and register spilling; yet if done well, it can provide orders of magnitude better performance in terms of run time and code size. Despite its importance, the Glasgow Haskell Compiler's inliner has not undergone significant change in decades.

This thesis explores the decision space of GHC's inliner; presents potential approaches to re-imagining the inliner, along with data and discussion for why they do not work; and additionally provides a conceptually simple analysis which guides inlining decisions at the function declaration site by advising the placement of compiler pragmas in source code.

The decision space exploration conducts an empirical investigation of the search space of GHC's inlining decisions as it pertains to the randomization of the magic numbers—or hand-coded constant integers—in the inlining heuristic. We demonstrate that randomizing the magic numbers produces some speedup over default GHC about half of the time, showing ample room for improvement. Then using iterative search, we produce four configurations into which we can cluster Haskell packages to produce a 26% mean speedup over 10 benchmark packages.

Following search space exploration, we investigate three machine learning models to predict inlining decisions: a genetic algorithm and neural networks from within the middle of the compiler, and graph neural networks to place pragmas at the source-code level. Our investigation reveals that although we can train the models with a high degree of accuracy, their predictions still fail to produce significant performance results when used in practice.

Finally, we devise a technique to place pragmas along hot call-chains—or complete, over-approximated chains of functions connected to profiled hot spots—which yields a 10% run-time speedup when combined with existing developer-placed pragmas, and 9% speedup without existing pragmas. This approach uses packages' abstract syntax trees to approximate control flow in quadratic time, whereas a conventional context-insensitive flow analysis would be cubic.

Lay Summary

Inline expansion, also known as inlining, replaces a function call site with the body of the called function during compilation. It has been claimed to be the single most important optimization in the Glasgow Haskell Compiler (GHC), and it is also the optimization that has received the most attention. Additionally, it is considered one of the hardest optimizations. Making bad inlining decisions may cause code bloat, cache misses, and register spilling; yet if done well, inlining can provide orders of magnitude better performance in terms of run time and code size. Despite its importance to performance, the Glasgow Haskell Compiler's inliner has not been significantly changed in decades.

This thesis explores changes to GHC's inlining decisions, presenting potential approaches to re-imagining the inliner along with data and discussion for why they do not work. Additionally, it provides a conceptually simple technique which guides inlining decisions by advising the placement of compiler pragmas, a type of manually added annotations, that allow developers to indicate to the compiler where to inline within the source code.

This thesis includes an empirical investigation of the search space of GHC's inlining decisions as it pertains to the randomization of the magic numbers—or hand-coded constant integers—within the compiler. We demonstrate that randomizing the inliner's magic numbers produces some speedup over default GHC about half of the time, suggesting ample room for improvement. Then using iterative search, we produce four configurations into which we can cluster Haskell packages to produce a 26% mean speedup over 10 benchmark packages. These benchmarks come from a framework introduced in the thesis, which produces them from real-world programs in the Haskell community.

After this search space exploration, we investigate three machine learning models to predict inlining decisions: a genetic algorithm and neural networks from within the middle of the compiler, and graph neural networks to place pragmas at the source-code level. Our investigation reveals that although we can train the models with a high degree of accuracy, their predictions still fail to produce significant performance results when used in practice.

Finally, we devise a technique to place pragmas along hot call-chains—a term we use to describe complete, over-approximated chains of functions connected to hot spots identified in code through profiling—which yields a 10% run-time speedup when placed alongside developers' existing pragmas, and a 9% speedup without the existing pragmas. This approach uses packages' abstract syntax trees (or tree representations of the program) to approximate control flow in quadratic time, whereas a conventional context-insensitive flow analysis would be cubic.

Acknowledgements

The lion's share of my gratitude goes to my advisor, Mike O'Boyle, who made this PhD possible in the most challenging circumstances. No words can convey my appreciation.

Significant additional thanks goes to Matt Might, who got me started; Simon Peyton Jones, who helped get me where I needed to be; Artem Pelenitsyn and Michael Ballantyne, good friends and generous colleagues who gave significant time to mentor other students, including myself; and Jan Vitek, for some advisorship along the way.

Thanks to Michel Steuwer for co-authoring my second Haskell paper and providing background knowledge on functional programming. I would also like to thank those whom I have had the privilege to work alongside and call friends in academia: Hyeyoung, Julia, Ming-Ho, Jordi, Elena, Alexi, Peter, Persie, Jackson, Steven, Pablo, Hugo, Akshar, Martynas, Ben, Brian, Jonathan, Alex, José, Tianyi, Josh, Krit, Ariel, Will, Pierce, Petr, Leif, and so many more whom I met at the University of Utah, Northeastern, and the University of Edinburgh—through labs, classes, and the Undistinguished Lecture Series.

I would like to thank my friend James Nagel for the extensive support at the start of my computer science career, personally and academically. I thank John Regehr and Suresh, professors who stayed in my life and gave me sass on social media—one of the best things that could happen post-masters degree. Additionally, I thank my friends in Edinburgh—Grace, Pavel, Dan, Pierre-André, and Dom—who encouraged me across the finish line.

Of course, I thank my family: Mom, Dad, Mark, Bev, Josh, Ben, and Mike.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Functional programming formulates programs in terms of pure functions, like mathematical functions, including higher-order functions which may be passed as first-class objects. This style of declarative programming minimizes side effects and produces deterministic behavior, along with modular code that is easier to reason about. Additionally, functional programming idiomatically encourages the use of immutable data structures over state change and the errors introduced by it. As a result of these qualities, functional languages produce software that is known for its reliability and low prevalence of errors.

Haskell is a purely functional programming language, based on the lambda calculus, with type inference and lazy evaluation. It is often regarded as a standard for functional programming languages. Haskell compiles ahead of time to native machine code, offering competitive performance against interpreted languages such as JavaScript, Python, and Ruby. To achieve this performance, Haskell relies upon the heavy use of optimizations during compilation. Among these optimizations, inlining is acknowledged to be among the most important. Additionally, it is acknowledged to be one of the trickiest.

Inlining is a well-known compiler optimization to reduce function-call overhead, where the compiler copies a called function's definition code directly into the function that is calling it. Within imperative programming languages, it removes the cost of pushing and popping registers to the system stack as well as enabling further downstream optimizations. In some cases, it can remove computed jumps—improving instruction cache behavior and, thus, execution time. Indiscriminate inlining, however, can cause excessive code size increase. Hence, most research focuses on how to benefit from inlining without incurring this cost.

Inlining has been studied for many years, and there is an extensive literature on its use for imperative and object-oriented languages [26, 66, 87, 156]. As functions are the foundation of functional programming languages, functional languages can benefit even further from inlining's reduction of function-call overheads and incurred allocations. Haskell is a higher-order polymorphic functional language where the treatment of functions, including inlining, dominates performance concerns. The Glasgow Haskell Compiler's (GHC's) inlining was once

said to be its single most important optimization [126], responsible for 20–40% of its performance gain [9, 139] (as opposed to 10–15% for imperative languages). Furthermore, the improvement of this optimization has been driven mostly by decades of hand tuning through trial and observation.

There has, however, been little further recent improvement. Although this may be partly due to the complexity of GHC's inlining heuristics [77], it is fundamentally a hard problem. The problem of finding an optimal inlining strategy is known to be NP-hard [141], with knock-on effects of subsequent optimizations exposed by inlining and an exponential inlining decision search space of 2^n, where n is the number of call sites which can be inlined [166]. As the number of call sites may be considerably higher in functional languages due to a convention of writing programs primarily through function composition, inlining optimization presents a significant challenge.

1.1 Inlining in the Glasgow Haskell Compiler

GHC has three parts: a front end for parsing, scope analysis, type inference, and desugaring to the intermediate *Core* language or *Core IR*; a middle part for performing Core-to-Core transformations; and a back end which translates the Core IR to Cmm (a small language similar to C--) which may then be passed to a choice of backends.

GHC's inliner is part of the optimizer in GHC's Simplifier, which resides in the middle part of the compiler [71]. The inliner replaces functions with their definitions in places where it expects a performance benefit, and it contains numerous safeguards to prevent inlining in detrimental cases—for example, functions which it deems too large (to avoid a performance penalty) or recursive functions where inlining might diverge.

When deciding whether to inline a function, GHC estimates whether the inlining will incur excessive code bloat by computing the function's size and then calculating and subtracting "discounts" which estimate how much of that code cost will be eliminated by further transformations [127]. The result of this calculation must fall beneath some threshold to trigger inlining. Altogether, the calculations which go into GHC's inlining decision process are known as its "inlining heuristics".

This thesis presents an exploration of various approaches to improve Haskell's inlining heuristics, including through machine learning and profile-driven analysis, whereas previous approaches have typically relied upon hand tuning with limited benchmarks. Following a layout of the technical background for the problem in Chapter 2 and related work in Chapter 3, Chapter 4 explores the efficacy of using random search and iterative compilation to improve

the existing architecture of GHC's inliner, Chapter 5 discusses attempts to apply machine learning to GHC's inlining decision framework and their resulting insights, and Chapter 6 presents a simple flow approximation to guide inlining pragmas closer to the source-code level to effect significant run-time speedup.

1.2 Contributions

This thesis presents a discussion on the challenges of inlining in the Glasgow Haskell Compiler based upon real-world Haskell code through the use of controlled experiments, demonstrates why inlining in functional programming languages is particularly hard to approach from the middle of the compiler where code has already been converted into Core intermediate representation, and presents a conceptually easy approach to direct inlining by over-approximating flow via profiling and then analysis of a program's abstract syntax tree. Although ample work has been done in the space of inlining for other languages, little has been done recently for Haskell. This thesis offers explanations for why machine-learning-based approaches fail, presents a practical solution in which over-approximation is allowable in the case of inlining, and posits a direction for further research.

1.2.1 Contributions of Chapter 4

Chapter 4 presents a benchmark framework which allows for the creation of benchmarks from real-world Haskell code in the Hackage Haskell package repository; conducts an in-depth experimental analysis of the performance of GHC's inliner across a range of real-world benchmarks; demonstrates with empirical evidence the benefits of using automated tuning techniques to improve the performance of the GHC inliner; and demonstrates the benefits of using a simple predictive model that delivers significant performance.

1.2.2 Contributions of Chapter 5

Drawing upon information from Chapter 4, Chapter 5 attempts to improve the inliner in the Glasgow Haskell Compiler through experimental use of three machine learning techniques: a genetic algorithm, neural networks in the middle section of the compiler, and graph neural networks to predict inlining pragma placement at function declarations in the source code. These experiments show that even when machine learning models are trained to high accuracy, performance improvements may not necessarily be seen. Additionally, reliable training data cannot be generated under the assumption that inlining decisions are independent of each other, such as measuring the effect of changing one inlining decision. Section 5.4.7 then provides a case study which shows that functions which are related by control flow can have a significant effect on performance when inlined altogether, which motivates the approach taken in Chapter 6.

1.2.3 Contributions of Chapter 6

Chapter 6 presents a profile-directed technique to recommend functions for inlining at the source-code level. Noting from Chapter 5 that giving inlining pragmas to entire call-chains of functions can effect a significant speedup in some real-world code, this chapter outlines an easy-to-understand method which approximates control flow via the abstract syntax tree so that inlining pragmas may be automatically added at function declaration sites. Experimental results show that even though this is an over-approximation of control flow, an overall 10% mean speedup can be observed when applying this technique to real-world Haskell packages. We see that there exists a subset of packages for which the technique provides a large speedup, yet the packages which produce no speedup remain virtually unchanged performance-wise; and furthermore, the technique produces a minimal change in code size, at just 1% mean size increase and less than 7% size increase for any individual package.

1.3 Structure

The organization of this thesis and summary of its chapters are as follows:

Chapter 2 discusses the technical knowledge which comprises the work and contributions presented in this thesis, along with an explanation of the methodology used for evaluation of its experiments. The technical background includes compilers, the Haskell language and ecosystem, benchmarking, profiling, and machine learning.

Chapter 3 presents work related to this thesis, which includes research on compiler optimization; inlining in imperative, object-oriented, and functional languages; and machine learning in optimization. As the work on compiler optimization is extensive, Chapter 3 presents samplings of work organized in terms of search-based approaches, predictive modeling approaches, and objectives.

Chapter 4 presents a benchmark suite for GHC based upon real-world packages from Hackage, the Haskell community's central open-source package archive; explores a compilation search space created from parameterizing and randomizing magic (hand-coded) numbers inside GHC's inlining heuristic at compile time to obtain optimal run-time speedups across 10 packages selected from Hackage; and introduces a simple model to cluster packages against 4 best-case magic number configurations based upon response time. This chapter is based on the work by Hollenbeck et al. [77], and it motivates further restructuring of GHC's inliner by empirically demonstrating that no single configuration of magic numbers approaches the maximum mean speedups observed across all packages.

Chapter 5 presents experimental work aiming to improve GHC's inlining heuristic from within the Simplifier and then at the source-code level through inlining pragma prediction. At the point of the Simplification pass, the work includes a genetic algorithm and artificial neural networks intended to predict non-trivial inlining decisions. At the source-code level, the work explores the use of graph neural networks to place `INLINE` or `INLINABLE` pragmas along function declarations with the use of training data obtained from both real-world examples of empirically tested "good" pragma-placement decisions and also experimentally produced examples of pragma-placement decisions based upon iterative search. This chapter explains why the aforementioned approaches failed to produce effective prediction models, motivating a strategy that leverages control flow in Chapter 6.

Chapter 6 introduces hot call-chains, a strategy whereby an over-approximation of control flow may be produced from a program's abstract syntax tree, to be used in combination with the placement of inlining pragmas at the source-code level along chains of connected functions (called "hot call-chains") to safely and effectively influence GHC's inlining decisions. This chapter motivates the technique with real-world code; outlines the experimental approach; presents the resultant speedups; and discusses the use of hot call-chains in combination with developer pragmas, along with a comparison of the use of either `INLINE` or `INLINABLE` pragmas along the hot call-chains. This chapter is based upon the work published in Hollenbeck and O'Boyle [78].

Chapter 7 provides a summary of contributions presented in this thesis, along with a critical analysis of the work done and an outline of future work.

Chapter 2

Technical Background

This chapter introduces some of the basics behind the technical concepts used in the work this thesis presents. Compiler optimization spans programming languages, machine learning, and software engineering; therefore, concepts are incorporated from each. Section 2.1 introduces the basics of compilers: overview, components, pragmas, and inlining as it pertains to Haskell in particular. Section 2.2 introduces Haskell—the language, the Haskell Cabal, the Hackage and Stackage package archives, and the Glasgow Haskell Compiler (GHC). Section 2.3 introduces benchmarking and how it relates to GHC. Section 2.4 discusses profiling, its implementation and how programs may be profiled with GHC and Cabal. Section 2.5 introduces machine learning: its basic terminology, supervised versus unsupervised and reinforcement learning, and the models used—neural networks, clustering, genetic algorithms, and graph neural networks. Finally, Section 2.6 gives the metrics and formulae to calculate the mean speedup.

2.1 Compilers

2.1.1 Compiler Architecture

What is a compiler?

A compiler is a program which translates source code into another form of code. Often, this translation is between high-level, human-readable code and machine-executable binary; however, other types of compilation may also occur. Compilers that convert high-level code to another high-level code are known as source-to-source compilers, transcompilers, or transpilers. Assemblers compile from assembly language to machine code, and disassemblers compile machine code into assembly language.

A compiler is typically composed of a front end, which takes in source code and produces an intermediate representation (IR); a middle, which performs optimizations over the IR; and a back end, which performs architecture-specific optimizations and code generation. These parts will be further described in this section.

This thesis is primarily concerned with the Glasgow Haskell Compiler (GHC), but both general compiler architecture and some specifics about GHC will be discussed in this section.

The Front End of a Compiler

The front end of a compiler converts source code into an abstract syntax tree (AST) to be passed to the middle of a compiler. The stages of this transformation typically include lexing, parsing, and some analyses and transformations. Figure 2.1 depicts these stages in their usual sequence.

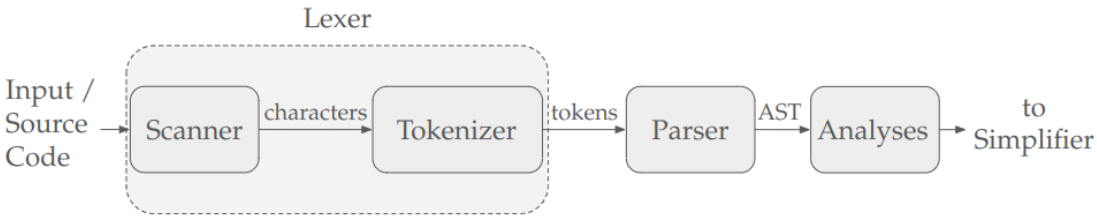


Figure 2.1: Stages in the front end of the Glasgow Haskell Compiler, simplified.

The lexer itself can be broken down into a scanner, which may include a tokenizer, which takes in a stream of characters and produces sequential tokens. For example, the input `2 + (3 * y)` might produce tokens like:

```
CONST(2) PLUS LPAREN CONST(3) MULT VAR(Y) RPAREN
```

After stripping the input of whitespace and comments, which have no effect on compilation, the tokenizer passes these tokens to the parser. From the lexer, the tokens are passed to the parser, which encodes them in a nested structure called an abstract syntax tree (AST). An AST represents the program in tree form.

In the Glasgow Haskell Compiler, the analysis passes in the front end include:

- **Renaming:** The Renamer resolves the scope of identifiers in the source files to make all identifier names unique and prevent clashes.
- **Typechecking:** The compiler ensures that the types of all inputs to operands are compatible with their operations; e.g., the program will not attempt to multiply an integer with a string.
- **Desugaring:** Code written in shorter human-readable syntax is converted into more verbose syntax which is better for machine execution.

The Middle of a Compiler

The middle of a compiler performs optimizations over the IR, taking in the IR and producing transformed IR, typically with the aim of making the resulting code smaller and/or faster. Often, these optimizations are independent of the targeted architecture. Some examples of such optimizations include but are not limited to:

Inlining: Discussed more in Section 2.1.3.

Dead-code elimination: Removes code which is determined to be unreachable or will have no effect on the program.

Constant propagation: Determines which variables have known constants at compile time.

Let-floating: Floating bindings from one place to another.

Strictness analysis: proving whether a function is strict in one or more of its arguments to enable the use of a more efficient calling convention.

Most of the optimizations in GHC occur in the middle of the compiler, also known as the Simplifier, which performs transformations over CoreIR. More information about GHC's middle and its IR, CoreIR, can be found in Sections 2.2.4 and 2.2.4, respectively.

The Back End of a Compiler

The back end of the compiler takes IR from the middle of the compiler after the architecture-independent optimizations have completed. It may additionally perform some optimizations for the target architecture, then outputs machine code targeted to a specific processor or operating system. This process typically includes register allocation and the generation of assembly code.

2.1.2 Compiler Pragmas

Compiler Directives

A compiler directive is a statement that a programmer may write into the source code of a program which will instruct the compiler on some aspect of how the code should be compiled. Some examples of directives in C include `#include`, which includes a file in the source code; `#define`, which indicates the definition of a macro; and `#ifdef`, which includes a section of code if a macro is defined using the `#define` directive.

Compiler Pragmas

Compiler pragmas are lines of code specific to individual compilers, rather than the grammars of languages themselves, which programmers may write into the source code to instruct a compiler on how to process and optimize certain input programs.

Compiler Optimization

When converting from one language to another—be it source code or a spoken human language—multiple interpretations of the same thing can be made which result in different translations. For example, idioms that exist in one language may not exist in the other, resulting in a variable selection of valid translations with differing qualities like brevity, precision, generalizability, etc. In the context of computing, programmers are most often concerned with two qualities of the result: its size and its execution time.

Optimizations of compilers typically aim to maximize some desirable quality—performance—by minimizing some undesirable quality: e.g., we want to minimize the size of generated code or its execution time to save on space or energy expenditure. This thesis specifically addresses the minimization of execution time in one compiler for one optimization: the Glasgow Haskell Compiler and the optimization of inlining.

2.1.3 Inlining

Overview

When a compiler inlines a function, it copies the code of the function directly from its definition into the site of the code calling it. This is done as an alternative to call linkage, where control is transferred to the target routine, entailing a passing of the function's parameters and the return of the function's value if applicable.

Too much inlining may cause code to get significantly larger, sometimes resulting in code bloat, which may slow down performance if the resulting code bloat causes thrashing. Alternatively, inlining may reduce the size of the resultant program even when a function is inlined numerous times. This is the case with very small functions, because the compiler may generate more code to handle registers and parameters than it would to simply inline the body of the function. It may also make execution faster by removing extra instructions associated with the call overhead.

It cannot be expected that inlining will make a difference for every program. In cases where performance is database-, network-, or I/O-bound, clever inlining may not offer much improvement.

Inlining in Haskell

In functional languages, inlining may simply be described as replacing the use of an identifier in an expression with the identifier's definition. An example in Haskell, originally presented by Peyton Jones and Marlow [126], is given below:

```
  let f = \x -> x*3 in f (a+b) - c
⟹  (a+b)*3 - c
```

Peyton Jones and Marlow [126] identify three distinct program transformations that collectively perform the inlining for the example above:

1. The *inlining* itself replacing a use of a `let`-bound identifier (here: `f`) by a copy of its definition (here: `\x -> x*3`):

   ```
     let f = \x -> x*3 in f (a+b) - c
   ⟹  let f = \x -> x*3 in (\x -> x*3) (a+b) - c
   ```

2. *Dead code elimination* that removes unnecessary `let`-bindings where the bound identifier is not used in the body of the `let`, as it is the case in the example:

   ```
     let f = \x -> x*3 in (\x -> x*3) (a+b) - c
   ⟹  (\x -> x*3) (a+b) - c
   ```

3. *β-reduction* transforming a lambda application into a `let`-binding, enabling further inlining:

   ```
     (\x -> x*3) (a+b) - c
   ⟹  (let x = a+b in x*3) - c
   ```

To finalize the example, we perform more *inlining* and *dead code elimination* steps:

```
  (let x = a+b in x*3)-c
⟹  (let x = a+b in (a+b)*3)-c
⟹  (a+b)*3-c
```

As Haskell is a lazy and pure functional language, *inlining*, *dead code elimination*, and *β-reduction* are always legal transformations that do not alter the program's meaning. *Dead code elimination* and *β-reduction* are easy to implement, as both of them are generally beneficial, whereas deciding when to inline what identifier is challenging. Therefore, GHC performs inlining with careful consideration, despite its heavy reliance on good inlining decisions for further optimization. To determine when inlining may expose further opportunities for optimization, GHC must examine the context in which the inlinee occurs to balance the benefits of inlining with potential negative effects, such as code duplication.

2.2 Haskell

2.2.1 The Haskell Programming Language

Haskell is a pure, general-purpose, functional, declarative, lazily evaluated language first released in 1990. It has garbage collection (versus static lifetime checking) and automatic memory management.

Haskell is statically typed, but type annotations are optional and inferred at compile time through bidirectional type checking. It compiles ahead of time directly to native machine code, unlike interpreted languages such as Python, Ruby, or JavaScript.

Additionally, Haskell has a rich type system which supports parametric polymorphism, algebraic data types, class-based polymorphism, runtime type inspection, existential quantification, type families, type equalities, higher-rank polymorphism, and kind polymorphism [183].

2.2.2 Hackage and Stackage

Hackage is the central package archive for Haskell [36]. All packages in Hackage are open source. Stackage is a set of Stable Hackage package sets, where eat set is a distribution of Haskell packages that are compatible and build with each other. Each of these sets is issued as a Stackage Nightly snapshot and then a Long Term Support (LTS) release.

Some terminology used in Haskell, Hackage, and Stackage include:

Module
A module in Haskell is a collection of code, under a namespace, in an environment of imports. A module contains declarations and expressions and may export its resources [70].

Program
A Haskell program is a set of modules.

Package
A package is a library of Haskell modules. Every Haskell program must define a Main module with a main function for a main package [160].

Dependency
A dependency is a package upon which a given source package relies, and it is indicated by a name and version or version range [158].

2.2.3 The Haskell Cabal

The Haskell Cabal (Cabal) [159] is a system for building and packaging Haskell programs. It contains both *cabal-the-tool*, which enables installation of Cabal, and *cabal-the-library*, which contains Cabal's code.

The Haskell Cabal Package Structure

The `.cabal` File
A package's `.cabal` file provides information about the package to the Cabal build system. Some of its fields include the executable name and main file, dependencies and their versions for building, whether the package is a library, its exposed modules, options for compilation, etc.

Tests
In a Cabal package, individual tests can be indicated with the `test-suite` field which includes a name for the test. Additional fields for the test indicate the location of its source folders and main file. This structure allows tests to be executed with the `cabal test` command.

Benchmarks
Benchmarks may be indicated in the Cabal package by the convention `benchmark` *name* and must have a name argument. A benchmark is a pure function or impure action that can be evaluated by comparison with a standard, e.g., for run time.

Commands
The command `new-build` will build every component of every package. It is run from the top-level directory. By default, Cabal will pass `-O2` to GHC for the level of optimization.

The -O2 Optimization Level
This is the default optimization level used by Cabal, and it can be described as "Apply every non-dangerous optimisation, even if it means significantly longer compile times." [163]

These optimizations, not included at lower optimization levels, include:

> **-fasm-shortcutting**
> Enable shortcutting at the assembly stage of the code generator. That is, if a block is only an unconditional jump, replace jumps to it by jumps to its successor.
>
> **-fdicts-strict**
> Make dictionaries strict, allowing the worker wrapper to fire on dictionary constraints. This often results in better run time.
>
> **-fspec-constr**
> Turn on call-pattern specialization to specialize recursive functions.

-fstg-lift-lams
Enable late lambda lifting on the Spineless Tagless G-machine (STG) intermediate language.

The command `new-test` *target* will run the given target(s) test suites. If the tests have not already been built by passing `-enable-tests`, then they will be built.

2.2.4 The Glasgow Haskell Compiler

The Glasgow Haskell Compiler (GHC) is a free and open-source compiler for Haskell. GHC supports concurrency, parallelism, profiling by time and allocation, and has a cross-platform software environment. It is Haskell's most popular compiler, and it is written in Haskell. Its runtime system, however, is written in C and C--.

CoreIR

GHC's IR, CoreIR, is an implementation of SystemF$_C$, which is System F extended with support for non-syntatic type equality. Its grammar includes metavariables, literals, term- and type-level variables, expressions, types, and coercions. Figure 2.1 presents the datatypes for SystemF$_C$'s expressions and variables, and Figure 2.2 presents its types [51].

e, u	$::=$		Expressions, *GHC/Core.hs*:Expr
	|	n	**Var**: Variable
	|	lit	**Lit**: Literal
	|	$e_1\ e_2$	**App**: Application
	|	**jump**$l\overline{u_i}^i$	**App**: Jump
	|	$\lambda n.e$	**Lam**: Abstraction
	|	**let** $binding$ **in** e	**Let**: Variable binding
	|	**join** $jbinding$ **in** e	**Let**: Join binding
	|	**case** e **as** n **return** tau **of** $\overline{alt_i}^i$	**Case**: Pattern match
	|	$e \triangleright \gamma$	**Cast**: Cast
	|	$e_{\{tick\}}$	**Tick**: Internal note
	|	τ	**Type**: Type
	|	γ	**Coercion**: Coercion

Table 2.1: Grammar for the datatype that represents expressions in SystemF$_C$.

GHC's Simplifier

GHC's middle, also known as the Simplifier, performs most of the source-to-source optimizations over CoreIR. Some of these optimizations include:

Inlining: Discussed more in Section 2.1.3. The process of inlining may expose the resulting IR to further and different optimization opportunities.

$\tau, \kappa, \sigma, \phi$	$::=$		Types/kinds, *GHC/Core/TyCo/Rep.hs*:Type
	|	n	**TyVarTy**: Variable
	|	$\tau_1\ \tau_2$	**AppTy**: Application
	|	$T\overline{\tau_i}^i$	**TyConApp**: Application of type constructor
	|	$\tau_1 \to \tau_2$	**FunTy**: Function
	|	$\forall n.\tau$	**ForAllTy**: Type and coercion polymorphism
	|	**lit**	**LitTy**: Type-level literal
	|	$\tau \triangleright \gamma$	**CastTy**: Kind cast
	|	γ	**CoercionTy**: Coercion used in type

Table 2.2: Grammar for the datatype that represents types in SystemF_C.

Demand analysis: In GHC, demand analysis is a form of strictness analysis, which analyzes the divergence properties of functions. This attempts to determine whether the functions may or may not diverge—which means to cause abnormal termination (such as failure with an error message) or loop infinitely—when given certain arguments. Demand analysis is a backward abstract interpretation approach to *strictness analysis*.

Rewriting with rules: Rewrite rules in Haskell allow developers to instruct GHC to rewrite code that matches a given pattern into a different specified form, using the `RULES` pragma with an optional phase-control number which specifies at what phase the rule should fire. This rewriting happens during the simplification pass.

Let-floating: Let-floating refers to the relocation of `let` or `letrec` bindings for performance improvement in heap allocation or execution time [124].

Constrained product result analysis: Determines when a function can profitably return multiple results in registers [15].

Specialization: GHC's specialization pass makes a monomorphic copy of every unique type a polymorphic function is called with, which removes some dynamic dispatches.

Constant folding: the process of recognizing and evaluating constant expressions with known values at compile time.

Beta reduction: is computing the result of applying a function to an expression.

The Guidance Data Type In GHC's Inliner

The `guidance` is a data type attached to expressions being considered for unfolding, and it carries some additional information to help guide that decision. Which type of unfolding guidance to attach to each expression is determined in GHC before occurrence analysis, which takes place before simplification; and the assignment of unfolding guidance may also be influenced by placement of inlining pragmas in source code. There are three types of unfolding `guidance`:

```
UnfNever
UnfWhen
UnfIfGoodArgs
```

UnfNever

If an item receives an `UnfNever` folding guidance, it is automatically excluded from consideration for inlining. The two situations in which an `UnfNever` guidance is given are: if the expression is calculated to have a size that is `TooBig`, or if the item is a top-level bottoming function. A bottoming function is a function that either fails due to error or goes into an infinite loop which returns nothing, and "top-level" means it is declared at the outermost level of a module.

UnfWhen

The UnfWhen data type constructor is shown below. As indicated in its comments, an expression is given a `guidance` of `UnfWhen` to disregard the size of the item when determining whether to inline. This is often the case with very small functions.

```
UnfWhen {   -- Inline without thinking about the *size* of the uf_tmpl
            -- Used (a) for small *and* cheap unfoldings
            --      (b) for INLINE functions
            -- See Note [INLINE for small functions] in CoreUnfold
ug_arity     :: Arity,    -- Number of value arguments expected
ug_unsat_ok  :: Bool,     -- True <=> ok to inline even if unsaturated
ug_boring_ok :: Bool      -- True <=> ok to inline even if the context
                          -- is boring
            -- So True,True means "always"
  }
```

UnfIfGoodArgs

The `UnfIfGoodArgs` guidance is given to all other functions whose RHS is neither too big nor small enough to warrant inlining without further consideration. Its data constructor is shown below.

```
UnfIfGoodArgs {     -- Arose from a normal Id; the info here is the
                    -- result of a simple analysis of the RHS

      ug_args ::  [Int],  -- Discount if the argument is evaluated.
                          -- (i.e., a simplification will definitely
                          -- be possible).  One elt of the list per
                          -- *value* arg.

      ug_size :: Int,     -- The "size" of the unfolding.

      ug_res :: Int       -- Scrutinee discount: the discount to
                          -- substract if the thing is in
    }                     -- a context (case (thing args) of ...),
                          -- (where there are the right number of
                          -- arguments.)
```

2.2.5 Inlining Pragmas in the Glasgow Haskell Compiler

Developers may manually add annotations to code, called *pragmas*, which give compiler-specific instructions for the build. Both the `INLINABLE` and `INLINE` pragma mark a function for more aggressive inlining and allow it to be specialized at use sites, even across modules. A *module* is a collection of functions, datatypes, classes, etc., defined together in the same namespace. Modules may import things from other modules when those things are marked as exported in their own modules. To inline functions from another module, however, it is necessary to have the *unfolding*, or the body, of the function available in an interface file. For very large functions, this will not happen unless they are explicitly annotated by an inlining pragma.

The `INLINABLE` pragma marks a function for inlining, where otherwise it may have been disqualified on account of its size, but allows GHC to make the final decision. An `INLINABLE` pragma would look something like this:

```
big_function :: Int -> Int
{-# INLINABLE big_function #-}
```

The INLINE pragma is stronger than INLINABLE. In comparison, the INLINE pragma makes GHC inline the function at every call site, as long as it is applied to at least as many arguments as there are on the left-hand side of its definition and inlining is safe (for example, not recursive). GHC will copy the unoptimized function definition into the interface file so that it can be used by externally defined functions. The INLINE pragma would look something like this:

```
big_function :: Int -> Int
{-# INLINE big_function #-}
```

It is still safe to put either inlining pragma on a recursive function: GHC can simply ignore it in that case [165]. In general, the INLINABLE pragma is considered safer to use for performance because GHC is allowed to make a cost-based judgment call on whether to inline it. The INLINE pragma, however, will largely override GHC's cost-benefit analysis—sometimes inlining things where size outweighs the potential benefits of inlining. Therefore, it is more likely to see a performance loss when placing an INLINE pragma in an ill-advised spot as opposed to INLINABLE [91].

2.3 Benchmarking

2.3.1 Overview

In computing, a benchmark is a test that measures the relative performance of the execution of a program or some other operation against a comparable reference evaluation. For the sake of this thesis, benchmarks are executions of Haskell programs with regard to the performance metric of wall clock time.

2.3.2 Benchmarking in the Glasgow Haskell Compiler

The Glasgow Haskell compiler group released a benchmark suite for lazy functional programming systems in 1993 called nofib [123]. These programs have since served as GHC's canonical benchmark suite, and they are still maintained to the date of this thesis. The nofib benchmark suite contains four types of single-threaded benchmarks, along with microbenchmarks for the -threaded runtime. Within the single-threaded benchmarks, the four categories are: imaginary, which are considered "toy" benchmarks, like puzzle solvers; spectral, which contains algorithmic kernels; real, which are command-line interface applications which the documentation describes as "rather aged"; and shootout, which are benchmarks from a benchmarks game called "language shootout". The nofib benchmarks have long since been considered out-of-date [105] but have not received a comparable replacement.

2.4 Profiling

Profiling measures characteristics of a program as it runs, such as its memory use, instruction activity, or time complexity. To do this, a program called a profiler either samples information from or inserts instrumentation into the program to be evaluated. However, because sampling has an element of randomness to its data collection, the information it provides is less complete.

2.4.1 Sampling

With sampling, the profiler examines the program's call stack at specified intervals—for example, after every millisecond—to collect information about the functions that are currently executing. This technique has very little effect on the execution of the application under examination.

2.4.2 Instrumentation

A profiler may insert hooks into the program's code or inject code into its binary which allows it to collect more precise information about it during its execution, such as the run time of each of its functions or their call count. The injection of these hooks, however, induces overhead.

2.4.3 Profiling in Haskell with Cabal

GHC will insert instrumentation into programs which can provide profile information for time and space consumption. Time measurements report CPU time, and space measurements include memory allocations. By default, profiling of a program's libraries is not enabled. When a program is profiled with GHC, only code written in the program will be instrumented. Blocking safe foreign calls will not be instrumented, but unsafe foreign calls will be instrumented. A safe foreign call is guaranteed to leave the Haskell system in a state that allows callbacks from external code. That is, the safe call accounts for the possibility of heap-allocated Haskell values to change to allow for garbage collection.

In profiling Haskell, the execution time or space of a program is its **cost**. Costs are attributed to **cost centres**, which are program annotations which enclose expressions. Cost centres may enclose other cost centres, which may produce a cost-centre stack to produce a call-tree of cost attributions.

At the time of this work, profiling through GHC may be run by building a program with profiling flags such as `-fprof-auto` or `-prof` and then running it with the `+RTS -p` options. An example of this, provided in the Glasgow Haskell Compiler 9.8.1 user guide illustrates how to profile a toy fibonacci program:

```
main = print (fib 30)
fib n = if n < 2 then 1 else fib (n-1) + fib (n-2)
```

To profile, the program is compiled accordingly:

```
$ ghc -prof -fprof-auto -rtsopts Main.hs
$ ./Main +RTS -p
121393
```

This produces output similar to the following in a `.prof` file:

```
    Wed Oct 12 16:14 2011 Time and Allocation Profiling Report  (Final)

       Main +RTS -p -RTS

    total time  =        0.68 secs   (34 ticks @ 20 ms)
    total alloc = 204,677,844 bytes  (excludes profiling overheads)

COST CENTRE MODULE  %time %alloc

fib         Main    100.0  100.0


COST                                          individiual   inherited
CENTRE   MODULE                no.  entries  %time %alloc  %time %alloc

MAIN     MAIN                  102        0    0.0    0.0  100.0  100.0
 CAF     GHC.IO.Handle.FD      128        0    0.0    0.0    0.0    0.0
 CAF     GHC.IO.Encoding.Iconv 120        0    0.0    0.0    0.0    0.0
 CAF     GHC.Conc.Signal       110        0    0.0    0.0    0.0    0.0
 CAF     Main                  108        0    0.0    0.0  100.0  100.0
  main   Main                  204        1    0.0    0.0  100.0  100.0
   fib   Main                  205  2692537  100.0  100.0  100.0  100.0
```

The second section gives a report of the “most costly functions in the program” [162] which may be referred to as a program’s “hotspots”. By default, the profiling report counts the most costly functions as those which consume 1% or more of either run time or allocations in the target program.

For the version of GHC used in this thesis, profiling instrumentation is inserted before optimizations. As of GHC 9.4.1, the option `-fprof-late` enables the compiler to add automatic cost centre annotations after optimizations.

2.5 Machine Learning

Machine learning is a collection of methods used to enable computers to solve problems via classification and decision making by inferring from patterns in data. Machine learning can be either *supervised* or *unsupervised*. Supervised learning uses labeled training data to "teach" models to classify, and unsupervised learning produces decisions from unlabeled data. Some commonly used terms are defined below for the subsections that follow.

2.5.1 Terminology

Model (machine learning)
A program which can make decisions or predictions about a dataset. In machine learning, the decisions may include (but are not limited to) labelling of two or more classes; regression, or estimation of the value of a dependent variable based upon one or more independent variables; similarity learning, or quantifying how similar objects are to each other; clustering, or discovering groups within a dataset; or dimensionality reduction, which is transforming data from a higher-dimensional space to a lower-dimensional space.

Training data
A set of examples used to train a model. For mathematical models, training data is often represented as an input vector of features and may be accompanied by an output vector (or scalar) which the model is supposed to learn.

Testing data
A set of examples which the model has not seen during training. The model is tasked to make predictions over the testing data, from which a metric of its expected accuracy may be ascertained.

Features
Features for machine learning are informative measurements of characteristics of examples of things over which the model learns and for which the model makes predictions. Most features are represented as numerical vectors or tensors. **Continuous features** are often referred to as *numeric*, and discrete-valued characteristics (arbitrary values which can be converted to "bins" of a certain numerical value, such as numbers corresponding to colors) are referred to as **categorical features**.

2.5.2 Supervised Learning

In supervised learning, machine learning models use labeled training data to learn how to make classifications or predictions over unseen testing data. Supervised learning includes active learning, classification, and regression. With supervised learning, an objective function for prediction is learned through iterative optimization—often stochastic gradient descent.

2.5.3 Reinforcement Learning

The basis of reinforcement learning involves an artificially intelligent agent taking actions in a dynamic environment to maximize one or more defined rewards. By learning patterns in data over a smaller number of observations, reinforcement learning aims to reduce search time for a solution and eliminate the need to generate a large number of data points for the alternative of supervised learning.

Reinforcement Learning may be modeled as a Markov decision process, where the definition of a Markov decision process has four components: S, a finite set of states; A, a finite set of actions; T, a transition function of the form $T : S \times A \times S \to [0, 1]$; and a reward function, $R : S \times A \times S \to \mathbb{R}$. [172]

One or more optimality criteria must be defined for the agent to try to maximize (or minimize) to learn a policy. One example would be the finite horizon model

$$E\left[\sum_{t=0}^{h} r_i\right], \tag{2.1}$$

where the agent wants to optimize the expected value of the reward r across time steps t for the known horizon number h. The criterion can be modified to, for example, give more value to later rewards by multiplying r by a discount factor γ^t where $0 \leq \gamma < 1$.

Feedforward Neural Networks

In feedforward neural networks (FFNNs), there do not exist any connections between the networks' units that form a cycle. that is to say, there are no connections that go backward to previous layers. Information may only travel forward in an FFNN from the input nodes to the output nodes.

Assuming an n-dimensional input, x, an FFNN computes a function f that approximates prediction(s) y:

$$f(x) \approx y \tag{2.2}$$

The training for the FFNN is a set of training data in pairs of (x, y). An FFNN may have zero to multiple hidden layers between its input nodes, which accept x, and output nodes, which emit y. Figure 2.3 depicts a feedforward neural network with four input nodes x_i, one hidden layer with two nodes h_i, and one output node y. The hidden layer nodes and output node may have activation functions. If we allow $v_i = \sum x_i + b$, where x_i is the value of the incoming i^{th} connection and b is a bias, then two historically common activation functions would be the sigmoids (σ) of $\tanh(v_i)$ (hyperbolic tangent function) and $(1 + e^{-v_i})^{-1}$ (standard logistic function). We can therefore re-write the FFNN equation as:

$$\sigma(\sum v_i + b) \approx y \tag{2.3}$$

where v_i is the input from the last hidden layer before the output node(s), which may itself contain an activation and multiple inputs.

More recently, rectified linear unit (ReLU) activations have been gaining popularity for their resistance to vanishing gradient problems and relative computational simplicity. Their activation is defined as:

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

Figure 2.2 depicts linear graphs for the inputs and outputs of the three activation functions described.

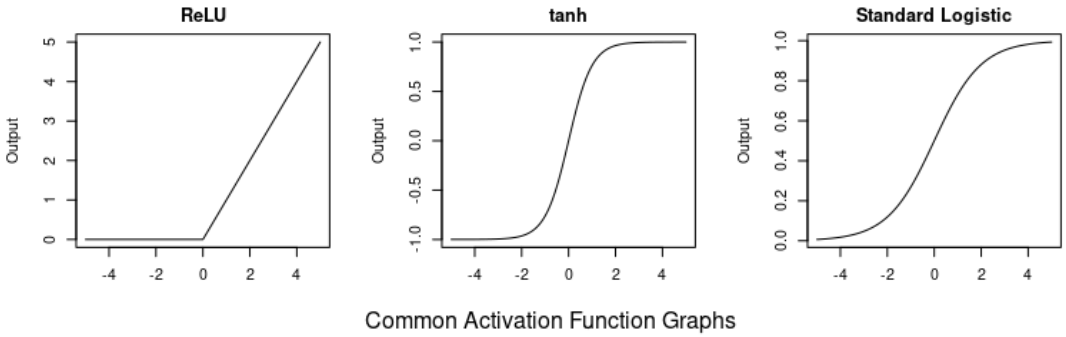


Figure 2.2: Three common activation functions for neural networks: the ReLU, the tanh, and the standard logistic function.

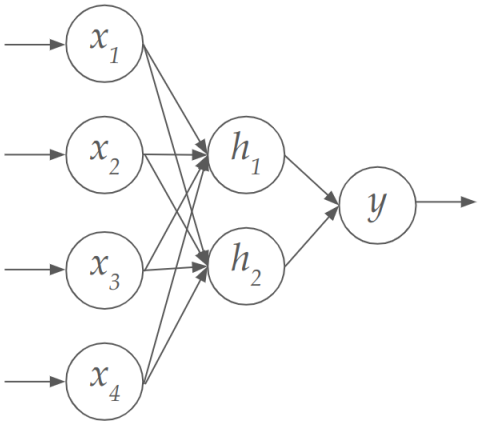


Figure 2.3: A feedforward neural network with 4 input nodes x_i, two nodes h_i in one hidden layer, and one output node y.

2.5.4 Unsupervised Learning

Unsupervised learning works with data sets that have no labels. With unsupervised learning, models infer patterns or associations in the data. For example, some unsupervised learning methods include clustering, or grouping data together by some attributes; anomaly detection, or identifying outliers in data; or latent variable learning, where observable variables are related to latent or hidden variables.

Clustering

Clustering, or cluster analysis, groups objects together by similarity of attributes. Some types of clustering include centroid-based clustering, or clusters where groups have a central vector and group membership is determined by which centroid each object is most similar to; connectivity-based clustering, where objects are placed in groups with other objects to which they are closest by some distance measure; and model-based clustering, where objects are assigned to distributions to which they are most likely to belong.

2.5.5 Feature Selection

Feature selection in machine learning is the process of choosing and cleaning a set of features to represent the problem situation for use with the model. Ideally, features should be relevant and help predict a correct output. Irrelevant features can slow down training or degrade predictive accuracy. Some common problems which arise from inadequate selection or representation of features may include

Multicollinearity

In the case of linear regression with ordinary least squares, when two or more predictive variables in a predictive model have a strong linear relationship—i.e., one variable can be written in terms of a linear function of the other—the output of the model may vary widely for small changes in its input. This happens because solving the least squares minimum problem $\min \|y - \mathbf{X}\beta\|^2$ may be approached by solving for the normal equation $(\mathbf{X}^\mathsf{T}\mathbf{X})\beta = \mathbf{X}^\mathsf{T}y$; and when sets of variables in $\mathbf{X}$ are multicollinear, the inverse of $\mathbf{X}^\mathsf{T}\mathbf{X}$ has an ill-conditioned inverse. That is, the approximate inverse may be incomputable or may have large rounding errors. Although stochastic gradient descent is resistant to multicollinearity, it may cause slower convergence.

Curse of dimensionality

The addition of each new feature adds a new dimension to the data set, which increases the space in which the data resides and hence requires more datapoints to avoid data sparsity. With the addition of more dimensions, datapoints appear increasingly dissimilar and farther apart, which makes learning more challenging.

Loss of information

Removal of informative features during data cleaning may result in the loss of signal which would be important for learning.

2.5.6 Genetic Algorithms

Genetic algorithms are a class of evolutionary algorithms, where evolutionary algorithms produce a population of individuals which evolve over some number of generations [125]. Within these populations, individuals are candidate solutions to a given optimization problem; and like the idea of Darwinian fitness, one or more of them selectively combine to produce new individuals for the next generation.

More specifically, genetic algorithms have the following steps and components, canonically [48]:

1. An objective function.
2. Candidate solutions represented as genomes, which contain all of the information necessary to produce a functioning version of the solution.
3. Random generation of a population(s).
4. Mating of two genomes via crossover, where genes within the genomes are inherited in the offspring, often with an element of probability.
5. Iteration of population production for either a fixed number of generations or until one or more fitness criteria are reached.

Genetic algorithms may be used to solve both constrained and unconstrained optimization problems—where constrained optimization problems have constraints on the system's variables and unconstrained optimization problems in which there are no such constraints. Convergence often takes numerous evaluations, and genetic algorithms may or may not converge onto a solution.

2.5.7 Convolutional Neural Networks

Overview

A convolutional neural network (CNN) is a deep neural network in which hidden layers perform convolutions with convolution kernels. The application of these kernels results in the production of feature maps which capture patterns of interest in the image. Examples of such features include shapes, edges, and texture.

Convolution

In mathematics, a convolution can be obtained by combining two functions, f and g, to form a third function, $(f * g)$. As a formula, this third function may be expressed as

$$(f * g)(t) := \int_{-\infty}^{\infty} f(\tau) g(t - \tau) d\tau, \tag{2.4}$$

where t denotes a time step or the position of a sliding window of input on the τ axis, where τ is a dummy variable after one of the functions has been reflected on the τ axis (e.g., $g(\tau) \rightarrow g(-\tau)$. The calculation of this integral allows a mirror of the function g to be "slid" over the function f to create a waveform representing the area under their intersection in "time".

Applications of convolutions include machine vision—for example, where convolution *kernels* in convolutional neural networks may use peaks in cross-correlations to indicate pattern matching; and for image processing, edge detection and blurring. Applications further removed from computing extend to physics, electrical engineering, fluid dynamics, and numerous other areas of science.

The Kernel Method

Classification in machine learning relies upon linear separability—or the ability to divide points in some N-dimensional space by slicing through it with a hyperplane, whereupon points on one side belong to one class and points on the other belong to another class. We denote a linear classifier parameterized by weights $\tilde{\boldsymbol{w}} \in \mathbb{R}^{n-1}$ and bias $b \in \mathbb{R}$, such that the classifier may be represented as

$$h(\tilde{\boldsymbol{x}}_{\boldsymbol{i}}) = \boldsymbol{\sigma}(\tilde{\boldsymbol{w}}^{\boldsymbol{T}}\tilde{\boldsymbol{x}}_{\boldsymbol{i}} + b). \tag{2.5}$$

In cases where the points are not linearly separable, no such hyperplane may exist to make the desired separation with $h(\tilde{\boldsymbol{x}}_{\boldsymbol{i}})$.

One trick to overcome this challenge is to map the points onto a space with more dimensions, creating one or more new axes upon which the desired linear separation can be made. Assuming feature vectors $\{\boldsymbol{x}_{\boldsymbol{i}}\}_{i=1}^{m}$ are not linearly correlated to fit their regression targets $\{y_i\}_{i=1}^{m}$, it may be possible to find an appropriate feature mapping $\boldsymbol{\phi}(\boldsymbol{x})$ such that the points are separable in a higher dimension.

The feature mapping, however, may require a much higher dimension than $\boldsymbol{x}$, making $\boldsymbol{\phi}(\boldsymbol{x})$ problematically costly to compute. Here, the Representer Theorem helpfully states that the problem of finding $\boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{\phi}(\boldsymbol{x})$ is the same as finding $\sum_{i=1}^{m}\alpha_i\boldsymbol{\phi}(\boldsymbol{x}_{\boldsymbol{i}})^{\boldsymbol{T}}\boldsymbol{\phi}(\boldsymbol{x})$.

2.5.8 Graph Neural Networks

Graph neural networks (GNNs) are neural networks which can operate over input structured as a graph, where graphs are objects containing information which can be represented as nodes linked by edges. These models can make predictions about any part of the graphs—including their nodes, edges, and the entirety of the graph.

Message Passing

Graph neural networks all use a form of neural message passing, where messages represented as vectors are passed between nodes in the graph [20]. Message passing allows inference over a graph in a manner which is permutation equivariant, meaning the output remains the same regardless of the node ordering or permutation of the rows and columns in its adjacency matrix [69].

Implementations of GNNs in this thesis use message passing layers formulated as

$$\mathbf{x}_i' = \gamma_\Theta\left(\mathbf{x}_i, \bigoplus_{j\in\mathcal{N}(i)} \phi_\Theta\left(\mathbf{x}_i, \mathbf{x}_j, \mathbf{e}_{j,i}\right)\right), \tag{2.6}$$

where $\bigoplus$ is a differentiable, permutation-invariant function like `max`, `mean`, `min`, `mul`, or `sum`, and γ_Θ and ϕ_Θ are differentiable functions such as multi-layer perceptrons [164]. The message passing scheme is a generalization of the convolution operator to irregular domains.

The graph neural networks models in this thesis use a convolutional operator of

$$\mathbf{X}' = \hat{\mathbf{D}}^{-1/2}\hat{\mathbf{A}}\hat{\mathbf{D}}^{-1/2}\mathbf{X}\Theta, \tag{2.7}$$

where $\hat{\mathbf{A}}$ is the adjacency matrix with self-inserted loops and $\hat{\mathbf{A}}$ is its diagonal degree matrix.

2.6 Evaluation Methodology

2.6.1 Metrics

Speedup

Calculation of speedup in execution time for the experiments presented in this thesis is done in terms of *relative performance*, where new execution times are compared to those of a baseline—or default execution time. The formula is thus:

$$speedup = \frac{t_{baseline}}{t_{new}}$$

Geometric Mean Speedup

The *geometric mean* calculates the average of a set of real numbers in terms of their product. The formulaic definition of the geometric mean may be presented as:

$$\left(\prod_{i=1}^{n} x_i\right)^{\frac{1}{n}} = \sqrt[n]{x_1 x_2 \cdots x_n}$$

This may alternatively be calculated as the arithmetic mean in logscale:

$$\exp\left(\frac{1}{n}\sum_{i=1}^{n} \ln a_i\right)$$

The geometric mean of a dataset where one or more values equals zero becomes zero itself, and these formulas are not meant to be used with negative numbers. Therefore, for the sake of the experiments in this thesis, the geometric mean speedup is calculated with respect to the speedup defined above, which will always be a positive non-zero ratio.

The geometric mean is a more appropriate measure than the arithmetic mean when calculating with performance ratios because the arithmetic mean does not give useful results for normalized numbers [53].

2.7 Summary

This chapter presented a basic introduction to the minimal knowledge required to understand the experiments which will be presented in Chapters 4, 5, and 6. This included an introduction to compilers, Haskell and its ecosystem, benchmarking, profiling, and machine learning.

Chapter 3

Related Work

This chapter presents the research done in relation to the contributions of this thesis. As the study of compiler optimization is extensive, related work can be organized and presented in a number of ways. Here, we present related work as an overview of compiler optimization in Section 3.1, including sections on traditional approaches and optimization with profiling; in Section 3.2, research related to inlining presented in chronological order for non-functional languages, followed by inlining in functional languages and Haskell specifically; and in Section 3.3, machine learning in compilation, broken down into search-based approaches, predictive modeling approaches, and objectives.

3.1 Compiler Optimization

The earliest example of compiler optimization may date back to FORTRAN, which promised to reduce personnel and machine debugging time through easy coding and fast execution in its 1954 preliminary report [39], which states:

> "Since FORTRAN should virtually eliminate coding and debugging, it should be possible to solve problems for less than half the cost that would be required without such a system."

Compilers do not, of course, eliminate coding and debugging; they merely lift it out of machine language and into source code. To enable this efficiently, about 25% of the instructions in the first FORTRAN compiler were written for optimization [142]. In the decades following, the field of compiler optimization has burgeoned, and it can be taxonomically discussed in a number of ways.

An early grouping of compiler optimizations divides them by global optimization versus local optimization. Global optimization considers the entire program, and local optimization considers one expression or statement at a time [**?**]. Another early grouping divides optimizations into three categories: *machine dependent*, where code execution is sensitive to the particular machines (e.g., register placement); *architecture dependent*, where performance may be tuned to hardware that is common between machines; and *architecture independent*, where gains can be made regardless of the computer system [142].

3.1.1 Traditional Approaches to Compiler Optimization

Machine-dependent Optimizations

As an early example of machine-dependent optimization, the Project for the Advancement of Coding Techniques (PACT) [110] compiler used a tabular set of rules tailored to the registers of the IBM 701 [142]. A generalized optimization still relevant today would be the *peephole optimization*, which takes place in post-processing. Peephole optimizations scrutinize the object code to find and rewrite inefficient sequences of instructions [106].

Initially, peephole optimizations corrected hand-written, machine-specific patterns; however, systems later generalized and pattern-matched these optimizations automatically. For example, Davidson and Fraser [43] match the emitted effects of symbolic simulations of sets of two- and three-instruction sequences against machine instruction descriptions, replacing instructions in instances where they can be reduced.

Architecture-dependent Optimizations

Early optimization work focusing on machines with common architecture included tree transformations in arithmetic expressions for instruction elimination in machines with common numbers of accumulators [6, 118]; instruction reordering [3] and dependency analysis for instruction scheduling [17, 167] for machines capable of executing parallel instructions. These architectures also prompted research into calculating and approaching maximum potential parallelization via, e.g., expression tree manipulation [129, 131, 154].

Architecture-independent/Global Optimization

Architecture-independent optimizations examine the entire program, often through flow analysis. Basic examples include constant propagation, which determines which variables will have known constants at compile time [54]; common subexpression elimination [35], where redundant computations are eliminated; and dead variable elimination, where unused variables are removed.

Constant propagation may be solved by determining a constant's *reaching definition* through a data-flow analysis which can determine statically where definitions may be reached within code [4]. Algorithms addressing this problem can often be used for dead code elimination, common subexpression elimination, elimination of redundant register loads, and live expression analysis as well.

A general solution by Kildall [88] uses a node traversal algorithm to keep track of changes in variable values at each basic block. Further work handles predicates and loops by evaluating conditional branches with constant operands [179, 180].

3.1.2 Use of Profiling in Compiler Optimization

Just-In-Time Compilation and Virtual Machines

Profile-guided optimization (PGO) in virtual machines, such as the JVM, use a combination of compiled and interpreted code and insert counters into code blocks which increment whenever it is used. This approach is known as *tiered compilation*, which has at least one *low tier* and *high tier*. When counters surpass a threshold, that code is designated as high tier and compiled with more optimizations [149]. Work in this area includes methods such as profile caching to persist profiling information across multiple application executions [12, 104].

Jantz and Kulkarni [83] explore single versus multi-level compilation, prioritization of method compilation, compilation options at various levels, single- versus multi-threaded compilers, and examination of many-core machines.

In .NET, the JIT compiler uses both tiered compilation and dynamic PGO [169], where the code's execution performance may change in the process of running a single instance of an application based upon information from that same run.

Profiling in Static Compilation

In static compilers, profiling information may sometimes be used to optimize compilation on a per-program basis. The GNU C++ compiler can optimize compilation based upon statistics collected from runs of its compiled programs. To enable this, programs may be compiled with special flags, run, and then recompiled with the generated statistics. This may be useful for branch prediction, loop unrolling, and loop peeling, among other things [56].

In 1992, Chang et al. [30] describe their implementation of a fully automatic inliner for C, which includes the use of a profiler to determine node (function) and arc (static function call) weights in call graphs. Benchmarks must be provided to obtain the profiling information.

A study by Grove et al. [65] showed experimentally through profiling that, for dynamic dispatch, receiver class distributions were strongly peaked and stable. Although the work was for C++, its results generalize to other languages with polymorphism.

Profiling in Functional Language Compilers

Bowman et al. [19] present a design for optimized profile-guided meta-programming in a meta-programming system and demonstrate its use with case expressions, receiver class prediction, and data structure recommendation. The system is described as general-purpose, but the authors implement examples in Chez Scheme and Racket, and they additionally suggest it for use in template Haskell, MetaOCaml, and Scala.

3.2 Inlining

3.2.1 Inlining in Imperative and Object-Oriented Languages

Before 2000

In 1988, Davidson and Holler [44] wrote a tool called INLINER which automatically inlined C modules. Although an instance was presented where inline expansion degraded performance, the size of the executable was otherwise shown not to cause performance problems. Kaser and Ramakrishnan [87] discuss power versus flexibility of inlining policies, where flexibility is the expressive capacity of the policy and power is the measure of a desired value of merit, and experimentally compares four different version-based inlining policies which do not consider additional specialization opportunities.

Deshpande and Somani [47] empirically study sets of C and C++ programs which show that C++ benefits more from inlining due to smaller function size and larger call stack depth. Also using empirical methods, Dean and Chambers [46] use inlining "trials" to capture the effects of inlining decisions and are stored in a persistent database for consultation in future decisions with similar call sites, using type group analysis, leading to performance gain.

One of the earliest experimental studies of inlining happened in Davidson and Holler [45], where the authors studied automatic inlining on a set of C programs and made some observations leading to improvements, including copy propagation on some parameters and the omission of local temporaries when a function's return value is used as a right-hand side of an assignment to a simple variable or when the function returns nothing.

Leupers and Marwedel [98] use a branch-and-bound algorithm to explore candidate inlinings while minimizing the number of dynamic function calls under a code size constraint, applying this method to C compilers for embedded processors.

2000 to 2010

Earlier work by Kulkarni et al. [93] use neuro-evolution to construct inlining heuristics for the Java HotSpot server compiler and the Maxine VM C1X compiler, then construct human-readable decision trees from these models.

For the X10 compiler, Alpern et al. [5] introduce *context-driven partial inlining* and *guarded partial inlining*. In these techniques, respectively, part of the called method is inlined based upon information available at the call site, or the frequently taken path through the callee is inlined and accompanied by a test and method to handle other paths.

Sewe et al. [144] predicts where nested inlining will occur in the Jikes RVM with information in programs' dynamic call graphs, to potentially eliminate guards in cases where call-sites are monomorphic. They do this by incentivizing the compiler to inline virtual calls if there exists a precise-induced or extant-induced edge targeting a method in the dynamic call graph. That is, the called method is of a precise type or, if extant-induced, a supertype in certain cases.

Lokuciejewski et al. [101] use random forests with features extracted from IR and the worst-case execution time (WCET) analyses to outperform standard inlining heuristics in the WCC C compiler against programs' WCET. Cavazos and O'Boyle [26] created inlining heuristics for the Jikes RVM compiler using a genetic algorithm trained from features of the caller and callee.

In Cooper et al. [37], an inliner for GCC is formulated as a condition string composed of clauses in disjunctive normal form which is applied in a postorder walk over expressions in the call graph, from the leaves to the root. Parameters in the condition are updated by a hillclimber algorithm with random descent to give a program-specific, adaptive inlining scheme.

Chakrabarti and Liu [29] introduce the concept of *inline specialization*, where a call site occurring in more than one call chain may be inlined selectively and in the most profitable version for each call site.

Zhao and Amaral [186] suggest an adaptive approach for the temperature threshold of the inlining heuristic, where temperature indicates cycle-heavy calling edges whose callee is small *relative to the program*. They posit that this adjustment of the temperature threshold on a per-program basis helps to prevent over-inlining in large benchmarks or insufficient inlining in small benchmarks. In Hazelwood and Grove [72], the authors use adaptive context-sensitive profiling to inline the appropriate versions of callees at each call site in the Jikes RVM.

Suganuma et al. [156] provide instrumentation in a Java JIT to dynamically collect call sites' distribution and invocation frequency and identify hot methods, as opposed to constructing a dynamic call graph through sampling during program execution, then use this information to hypothetically demonstrate that profiling information may be effective for improving compilation or performance.

2010 to the Present

In the last ten years, Theodoridis et al. [166] present work which primarily gives an exhaustive empirical analysis of inlining policies for binary size reduction over the SPEC2017 benchmark suite with LLVM, but they also present a simple autotuning strategy. The autotuner performs parallel compilations where the edges of the call graph are inlined and checked for size reduction. Results of previous rounds are then used as initial inputs to subsequent rounds.

For Java's Jalapeño dynamic optimizing compiler, Arnold et al. [11] use a combination of static and profile-based heuristics for inlining using static and dynamic call graphs with weighted edges and nodes. Profiling information is represented as weighted nodes, where nodes represent procedures or methods and their weights represent the dynamic frequency of their calls.

Prokopec et al. [130] present an algorithm for JIT compilers that uses adaptive decision thresholds, callsite clustering, and deep inlining trials. Callsite clustering refers to identifying related callsites, and deep inlining trials are performed by propagating callsites' argument types throughout the call tree and optimizing the entire call tree to estimate performance savings.

To reduce compilation time for Java as it is JIT-compiled, Ochoa et al. [120] trade off a significant increase in code size and minor increase in execution time. They do this by using abstract interpretation to compute reusable method summaries of inlining candidates' potential optimizations.

In 2021, an artificial neural network (ANN) is used to predict resultant code size after code duplication passes in GraalVM, where the feature vectors are comprised of counters for the node types in the Graal IR. Over several benchmarks, code size for the ANN ranged from 3% to 25% larger than the default GraalVM, and run time ranged from 3% slower to 14% faster; however, only one benchmark was over 3% faster than default.

3.2.2 Optimization of Inlining in Functional Languages

Appel [10] introduced "loop headers", implemented in an intermediate language in Standard ML, to distinguish between recursive calls and outside calls to the function. Loop headers enable invariant arguments to be lifted out of loops and provide a binding site for variables.

Jagannathan and Wright [81] implemented an inlining optimizer for R^4RS Scheme based upon a polyvariant control-flow analysis [21, 86]; however, even a simple context-insensitive control-flow analysis is cubic in complexity and difficult to implement in practice [63].

Heintze and McAllester [74] present a linear-time subtransitive CFA which calculates a full transitive closure of a call graph in quadratic time; however, the algorithm does not handle untyped or recursively typed programs.

In a functional language versus an imperative language, functional representations of code create more closure allocations; or in other words, they allow more opportunities to eliminate allocations via inlining. Significant work has been done to expose opportunities to eliminate these allocations in OCaml [111].

The MLton compiler uses *whole-code optimization* for Standard ML, which uses “defunctorization”, “duplication”, and “defunctionalization” [147] which reduce modules, polymorphic data types and functions, and higher-order functions into first-order code based upon look-up data structures. This transforms the code into a simply typed, first-order intermediate language (IL) which can be combined with aggressive inlining and dead code elimination.

Danvy and Schultz [41] describe lambda-dropping, in which recursive programs are given blocks with lexical scope. This then allows blocks to float and allows for further reasoning about and optimization of the program. Lambda-dropping is presented as a symmetric transformation to lambda-lifting [85], in which block-structured programs are turned into recursive equations while preserving meaning.

Work by Serrano [143] lays a framework for inlining recursive functions in which the inlining decisions is based upon the size of the function, the size of the call, and the location of the call. The algorithm, demonstrated in Scheme, allows code growth by a specified factor for each call site; and when the inlining occurs, the algorithm recurs and reduces the factor.

Inlining Optimization in Haskell

Most approaches to compiler inlining optimization target imperative programs where the control-flow graph and function parameter types are statically known. Haskell is a higher-order polymorphic functional language where the treatment of functions, including inlining, dominates performance concerns.

The most influential work for inlining in Haskell, and other functional programming languages, are the key lessons presented by Peyton Jones and Marlow [126] from work on the Glasgow Haskell compiler’s (GHC’s) inliner. In this work, the authors confirm that GHC’s inlining is both the single most important optimization and that its improvement is made through ten years of hand tuning through trial and observation. They present an algorithm to inline recursive functions—namely identifying a *loop breaker* to prevent non-termination, identifying conditions for when inlining should occur in GHC, and how to track lexical and dynamic environments to enable additional transformations when the state of free variables becomes known at the inlining site. For hand tuning, GHC has a canonical benchmark suite called NoFib [123].

3.3 Machine Learning in Compilation

3.3.1 Search-based Approaches

Iterative Compilation

The process of iterative compilation involves generating a number of different versions of a program and comparing them, often according to execution time or binary size. Some of its early applications include work for embedded systems and kernel benchmarks. Using a simple iterative search algorithm, Bodin et al. [18] come within 0.3% of optimal execution time while visiting less than 1% of the total search space—demonstrating a use case for embedded systems, although the approach is too expensive for general computing. Kisuki et al. [90] created multiple compilations using four search algorithms to find loop unroll factors and tile sizes over three sets of small kernel benchmarks from multimedia applications, using the Fortran77 compiler.

Work by Agakov et al. [2] later proposed to reduce the search space of this approach by learning probabilities of improvement for transformations, given the prior sequences of transformations, by representing sequential transformations as Markov chains. Sequences and transformations that yielded performance gains were used as initial populations in a genetic algorithm (GA), trained over C programs, which was evolved to make predictions for further transformations.

For LLVM, Ganser et al. [58] focus on parallelization and tiling within the polyhedra space, showing that a simple random search of configurations can deliver performance without the need for more expensive technologies such as genetic algorithms.

In Chen et al. [32], the authors demonstrate that iterative optimization may be largely data-insensitive by collecting 1000 data sets for 32 programs using Intel's ICC and GNU's GCC and finding at least one set of compiler optimizations that achieves 86% or greater speedup across all data sets for each.

Auto-Tuning

Auto-tuning of compilers primarily focuses on the automatic selection of optimizations and the phase-ordering of those optimizations [13]. Machine learning has led to contributions in auto-tuning through the use of, but not limited to: search space exploration, deep neural networks, and Bayesian methods.

Compilers for deep learning, which are specialized to optimize models trained by deep learning frameworks, have received a large amount of attention in only the last few years. Recent work in Tollenaere et al. [168] defines a structured configuration space for faster convergence onto loop transformations for tensor computations, compared to previous auto-tuning code

generators, focusing on two-dimensional convolutions on CPUs. Also for exploring transformations to tensor programs to target hardware, Gibson and Cano [62] introduce transfer-tuning, where similar auto-schedules may be identified and reused between programs, and demonstrating the technique on pre-tuned DNNs to new DNNs.

Work by Ryu et al. [135] aims to predict optimized tensor operation codes without repeated search and hardware measurements by using a transformer-based predictor. Zhang et al. [184] use Bayesian Upper Confidence Bounds (UCB) to predict the performance gains of tensor operators with uncertainty quantification.

Advancements with auto-tuning in compilers for other fields of computing include Liu et al. [99], which aims to solve multitask optimization of application code for exascale and Message Passing Interface (MPI) applications, using Gaussian processes in Bayesian optimization.

Similarly, Hellsten et al. [75] reason about permutation, ordered, and continuous parameter types with known and unknown parameter constraints for compiler systems targeted towards CPUs, GPUs, and FPGAs using Bayesian optimization. Also for porting code across different platforms, Ashouri et al. [14] present an autotuning framework based upon Bayesian networks and independent microarchitecture features.

For multicore autotuning, Ganapathi et al. [57] use kernel canonical correlation analysis (KCCA) to find multivariate correlations between vectors of configuration parameters and performance metrics, demonstrating this technique over stencil code optimizations with features including thread count, software prefetching, padding, etc.

Ansel et al. [7] present OpenTuner, a general framework to build domain-specific multi-objective program autotuners; employ search technique ensembles for tuning; and demonstrate the framework's use on small and large search spaces exceeding 10^{3600} configurations.

Training Data Generation

Generation of labeled training data for machine learning for compilation can become costly due to compilation time combined with the search space of numerous exposed and unexposed heuristics. In GraalVM, Mosaner et al. [113] propose compilation forking, where each potential heuristic parameter is forked during compilation and executed with n parameter values to be explored, producing datapoints to compare local optimization decisions.

Another method to generate data for ML models used in compilation is multi-versioning [33, 96], where multiple versions of code are generated on the whole-program or partial-program level (e.g., function level) [31, 61, 189].

Automated Feature Extraction

Namolaru et al. [119] propose a method to systematically generate numerical features from a program using Datalog representations of binary relations encoded in subgraphs produced during compilation, such as control flow, def-use chains, and intermediate representation. An example of such a feature would be taking an aggregate of the number of store instructions in basic blocks to determine the average number of stores per basic block. They propose the use of domain knowledge to incrementally add features for use in a predictive model and evaluate the technique on an independent identically distributed (IID) model and a Markov model, per those used in [2], to select sets of optimizations among 88 in GCC.

Heuristic Discovery/Creation

Mosaner et al. [114] use the compilation forking method from Mosaner et al. [113] to produce training data over code features to automatically create heuristics by tuning neural networks at run time in a dynamic compiler, using loop peeling as an example.

Leather et al. [97] use genetic algorithms to explore features over program IR and incrementally build an optimization feature set through a greedy approach, demonstrating their system on loop unrolling in GCC. For targeting multiple architectures, Saha et al. [138] present a framework which provides and trains models over training data which it generates on the targeted platform, providing an abstracted interface for a set of common modeling and tuning problems and demonstrating it on register allocation for GPU kernels. Their autotuner produces multiple variations of code for training data from a small number of samples by varying parameters such as tiling, unrolling, optimization flags, and thread geometry.

Option Selection

For ordering compiler optimizations, also known as phase ordering, Kulkarni and Cavazos [92] use neuro-evolution to construct an artificial neural network which predicts orderings on a per-program basis, implemented for the Jikes RVM. Zhong et al. [187] use simulated annealing to tune optimization options in the GNU Compiler Collection.

3.3.2 Predictive Modeling Approaches

Compiler optimizations may be grouped by choice of predictive models. Some examples of such modeling decisions are presented as follows.

Regression

Linear regression is a statistical model that may be considered a simple form of machine learning. Dubach et al. [50] use linear regression to predict speedups in code execution time based off of code features transformed into a smaller feature vector with principal component analysis (PCA).

Jiang et al. [84] identify correlations of program component behaviors—e.g., trip counts between loops—to inform behavior prediction for optimization. They introduce *seminal behaviors*, a core set of behaviors that strongly correlate with other program behaviors and reveal themselves within the initial 10% of execution time, and calculate these behaviors statistically via the Pearson product-moment correlation coefficient. The authors then use least mean squares linear regression and regression trees as predictive models, demonstrating the technique in profile-directed-feedback compilation in the IBM XL compiler.

Logistic Regression, a simple method for supervised learning, may be used to predict two or more categorical values as output. For example, Cavazos et al. [24] predict optimal vector masks of compiler option recommendations using logistic regression with an input of values from 60 performance counters. Cavazos and O'Boyle [27] predict which optimization set to use in the Jikes RVM JIT compiler on a per-method basis using logistic regression over bytecode features.

Polynomial regression, unlike linear regression, can fit non-linear relationships between variables. In Luo et al. [102], the authors use polynomial regression to predict the similarity of feature vectors against those in a database associated with optimal solution spaces—or a set of optimal solutions—for stencil autotuning.

Support Vector Machines

Wen et al. [181] schedule multiple program kernels on CPU/GPU heterogeneous platforms using static code features with a support vector machine (SVM). The training data and test data are OpenCL kernels, and the SVM predicts a large speedup or not.

To predict optimization sequences for runtime speedup, Park et al. [122] extract a graph-based characterization of programs' control-flow graphs from intermediate representation for use with SVMs combined with graph-based kernels including the shortest path graph kernel, Gaussian kernel, and Brownian bridge kernel.

For auto-parallelization, Wang et al. [178] use profiling-based dependence analysis to identify parallelization candidates and SVMs to map decisions. They demonstrate this approach on the NAS and SPEC CPU2000 benchmarks over two multicore platforms.

Taylor et al. [157] use SVMs to map OpenCL kernels onto embedded heterogeneous multi-core platforms and choose the processor frequency, and the approach is adaptable to different optimization goals. Also for parallelization, Zhang et al. [185] use SVMs to predict resource partition and task granularity, evaluated on a CPU-XeonPhi mixed heterogeneous many-core platform.

Artificial Neural Networks

Singh et al. [148] compare multiple linear regression with ANNs for predicting parallel application scalability and find that the linear regression has slightly better performance and accuracy; however, the study uses a simple 3-layer feed-forward neural network. In addition to linear regression, Dubach et al. [50] use ANNs to predict run-time speedups by code features.

After identifying 6 program features, Yuki et al. [182] generate synthetic programs from these features to use as training data for an ANN to predict tile size selection models. Taking an end-to-end approach, Cummins et al. [40] train deep neural networks over raw source code to predict optimal mapping for heterogeneous parallelism and GPU thread coarsening factors for OpenCL.

Decision Trees

A decision tree is a structure which may be represented as a flowchart of tests upon attributes of the item being considered for classification, and it may be produced from supervised training data or in an unsupervised manner using techniques such as clustering [175] or genetic programming [22]. Decision trees have been effectively used to generate optimization heuristics such as those used in loop unrolling [112] and to automatically generate OpenCL code for GPUs from data-parallel OpenMP programs [64]. Algorithms may also combine the decisions from multiple decision trees into a random forest. In Benedict et al. [16], the authors use random forest modeling to predict energy consumption of OpenMP applications in compilers.

Online Learning

Online learning aims to train models over incremental data points, rather than batch learning which learns over large batches of data points at a time—which often scale poorly in real-world problems with evolving data. Many online learning problems may be formulated as an Online Convex Optimization problem [76]. Compilation problems are often formulated in the context of online learning due to the cost of computing each data point (e.g., a compilation instance of a program or benchmark suite).

Reinforcement Learning

For compilers, reinforcement learning may be used where creation of labeled training data is difficult or impossible, such as learning the relationship between IR codes and effective optimization strategies [145], determining high-level synthesis phase ordering to create digital hardware circuits [68], and optimizing inlining [170].

Bayesian Optimization

Bayesian optimization has been demonstrably useful to estimate auto-tuning parameters in compilers, such as unrolling factors, tiling factors, parallelization schemes, and autoscheduling [75]. Although Bayesian optimization was initially used for estimations over parameters with continuous compact domains, research suggests they may be a good choice for discrete domains as well [60].

Evolution-based Algorithms

Although genetic algorithms are not themselves predictive models, they are used to create and tune predictive models. Cooper et al. [38] presented an early genetic algorithm to determine ordering of optimization sequences to optimize for code size for C and FORTRAN. Ansel et al. [8] use a genetic tuning algorithm to generate candidate populations of compiled algorithms for variable-accuracy domains such as signal processing or NP-hard problems which allow approximate solutions.

For a variety of compiler heuristics, Stephenson et al. [153] use evolutionary algorithms to search the solution space of priority functions, which are functions that prioritize options for a compiler algorithm—for example, list scheduling, data prefetching, and register allocation.

Also for multiple optimizations, Hoste and Eeckhout [79] search the Pareto frontier of individual optimizations in all compilations levels of the GNU Compiler Collection (e.g., -O1, -O2, etc.) that can be toggled, using a multi-objective search algorithm based the Strength Pareto Evolutionary Algorithm [190, 191].

Garciarena and Santana [59] use estimation of distribution algorithms (EDAs) including a genetic algorithm, a univariate marginal distribution algorithm, and a dependency-level EDA to learn probabilistic models of solutions that learn patterns among the solutions—including interactions between variables (i.e., compiler flags).

Cascaded Prediction

Cascaded prediction uses predictors arranged in two or more levels. For example, Driesen and Holzle [49] use a two-level cascaded approach for branch predictions: the first predictor handles inexpensive predictions, and the second predictor makes more computation-intensive decisions if the first predictor fails.

Involving machine learning, Magni et al. [103] use a cascaded neural network approach to iteratively query the model at every opportunity for thread coarsening—across new programs and upon the same program after a previous thread coarsening, as repeated coarsening may have beneficial or deleterious effects for performance.

Clustering

Sherwood et al. [146] present a cluster-based analysis of large-scale program behavior to identify program subset simulation points for computer architecture research. They create basic block vectors—or 1-dimensional arrays representing static blocks in the program—which may potentially have millions of dimensions, use random linear projection to project the data onto a 15-dimensional space, then cluster the data points with k-means using Euclidean and Manhattan distance measures. Centroids are considered representatives for each cluster.

For streaming parallelism to multi-core processors, Wang and O'Boyle [177] predict ideal partition structure with the nearest neighbor classifier by evaluating randomly generated partitions represented by 10 features including pipeline depth, branches per instruction, and maximum dynamic rate. They demonstrate this approach over StreamIt applications with the StreamIt compiler.

For performance prediction, Hoste et al. [80] transform a set of microarchitecture-independent characteristics—information relating to the instruction mix, ILP, register traffic, working set size, etc.—into spaces in which they can compute relative distance measurements against a set of benchmark programs. They compare normalization, principal component analysis, and a genetic algorithm for computing weight assignments as techniques to produce a data transformation matrix which maps programs of interest into a new space; and in this new space, the speedup prediction is a weighted harmonic average over the speedups of benchmarks, called "proxies", near the position of the given application.

3.3.3 Objectives

Tasks at every level of the compiler may be targeted for improvement with machine-learning-driven optimization. Some examples include but are not limited to: for the front end, CPU and GPU partitioning; for the middle, phase ordering, flag selection, and vectorization; and or the back end, register allocation and instruction scheduling.

High-level Optimizations

Saroliya et al. [140] use deep Q-learning for hierarchical partitioning on GPUs by targeting both co-scheduling and resource partitioning with an accompanying co-scheduling and resource management system which employs offline training with online optimization. Saba et al. [136] formulate a system-wide optimization problem to determine co-scheduling decisions, resource partitioning, and power capping. They use a simple feed-forward neural network over the inputs of hardware configuration and performance counters collected from the CPU and GPU during a profile run for jobs in a set to predict a slowdowns and the affect of power capping. Additionally they use Edmonds' algorithm for finding a spanning arborescence of minimum weight to optimize the hardware configuration and minimize the execution time of co-scheduled jobs.

Middle Optimizations

Stephenson and Amarasinghe [152] use two supervised learning techniques, near neighbor classification and support vector machines, to predict loop-unrolling factors using the Open Research Compiler over the SPEC 2000 benchmark suite.

For auto-vectorization, Mendis et al. [107] propose a strategy for learning vectorization similar to the method used by [95], where the decision procedure is formulated as a Markov Decision Process for use with the DAGGER algorithm [134] to collect traces from the Integer-Linear Programming solver over the vectorization problem, after which the traces are used as training data for a Gated Graph Neural Network for a parameterized policy. In [67], an end-to-end vectorization approach entails feeding benchmark source code into a loop extractor to produce loop code segments which are then fed to en embedding generator, after which the learned embeddings are input into a Deep Reinforcement Learning (RL) agent which dynamically determines the loops' vectorization factors. Jain et al. [82] address loop distribution for vectorization and locality by generating a strongly connected component (SCC) dependence graph for each loop and feeding them to an RL model that determines their distribution order by performing topological walks over the graphs.

Low-level Optimizations

Wang and O'Boyle [176] used predictive modeling to map program parallelization for OpenMP programs to Intel Xeon and Cell processors, using feed-forward neural networks to predict scheduling and support vector machines for scheduling policy classification.

Very recently, the problem of register allocation has been addressed as a graph-coloring problem [28] with reinforcement learning. VenkataKeerthy et al. [174] use an embedded multi-agent hierarchical reinforcement learning (RL) algorithm in LLVM Register allocation schemes based upon machine learning have semantic constraints which need consideration; e.g., register types must be considered and variables in the same live range must not be assigned to the same register. In RL, these constraints may be imposed in the action space. Das et al. [42] use a deep learning network based upon several layers of LSTM which output a color for each node of graph. Because the network may allocate the same color, or register, to nodes connected by an edge—meaning multiple results have been assigned to the same register—the authors augment the network with a color correction phase.

Work for instruction scheduling includes Eliot et al. [52], where the order of instructions in basic blocks are claimed to be scheduled near-optimally in the SPEC95 benchmark of FORTRAN and C code. The authors use a small variety of supervised learning techniques, including decision trees and feed-forward neural network, with only five features. Cavazos and Moss [25] use supervised learning with induced heuristics to predict scheduling benefits for basic blocks in the Jikes RVM. In this work, the authors learn rule set induction over block features to determine when to schedule blocks, as scheduling can sometimes degrade performance.

Architecture-specific Optimizations

Cavazos et al. [23] train feed-forward neural networks (FFNNs) in a supervised manner over a vector of transformation sequences with observed speedups as response to predict execution time on any given architecture, with the motivation that sometimes architectures can only be evaluated on a simulator—which can take more time than running on the actual processor. Similarly, Nadeem et al. [117] use FFNNs to predict the behavior of chip-multiprocessor design configurations in regard to energy-delay and execution time using "reaction-based modeling", where predictions are based off of observations of program behavior on a subset of architecture configurations. They evaluate their approach on both explicitly parallel and thread-level speculation applications.

Vaswani et al. [173] use micro-architecture sensitive models empirically, including the use of radial basis function networks (a type of neural network), parameterized by binary and numeric compiler flags, heuristics, and micro-architectural parameters to predict an execution time response. Additionally, they use genetic algorithms based upon empirical models frozen at platform-specific settings to explore near-optimal solutions.

For GPUs, Liu et al. [100] use cross-input models predicting parameterized values for optimizations. They first take an iterative empirical approach that caches attempted predictions made by a hill climber in a performance database, then the information in the database is processed to find relations between inputs and optimization decisions through the use of regression trees.

3.4 Summary

This chapter discussed the related work pertaining to the topics presented in this thesis, including multiple ways of organizing the research related to compiler optimizations. Although several advancements have been explored in this chapter in relation to inlining, less previous work exists for inlining in functional languages—with very little work for inlining in Haskell, specifically.

Chapter 4

Investigating Magic Numbers: Improving the Inlining Heuristic in the Glasgow Haskell Compiler

In this chapter, we present an in-depth study of the effect of inlining on performance in functional languages. We specifically focus on the inlining behavior of GHC and present techniques to systematically explore the space of possible magic number values, or configurations of magic numbers, where magic numbers are hand-coded numeric values written into the source code of GHC's inlining heuristic. We evaluate the performance of these magic number configurations on a set of real-world benchmarks where inline pragmas are present. Pragmas may slow down individual programs, but on average they improve performance by 10%. Searching for the best configuration of magic numbers on a per-program basis increases this performance to an average of 27%. Searching for the best configuration for each program is, however, expensive and unrealistic, requiring repeated compilation and execution.

This chapter determines a new single configuration of magic numbers that gives a 22% improvement on average across the benchmarks. Finally, we use a simple machine learning model that predicts the best configuration on a per-program basis, giving a 26% average improvement.

Contributions from this chapter include a benchmark framework which allows the creation of benchmarks from real-world Haskell code in the Hackage Haskell package repository; an in-depth experimental analysis of the performance of GHC's inliner across a range of real-world benchmarks; a demonstration with empirical evidence for the benefits of using automated tuning techniques to improve the performance of the GHC inliner; and a demonstration of the benefits of using a simple predictive model that delivers significant performance.

Section 4.1 provides an overview of GHC and its inliner, with a focus on magic numbers inside the inlining heuristic. The approach of the experiments are described in Section 4.2, including the modified compilation parameters, constructed benchmark framework, and benchmark selection. Section 4.3 outlines the experimental setup and Section 4.4 the experimental results.

The experiments and their results are then analyzed in Section 4.5, which also discusses best configurations of magic numbers across the benchmarks and how well the approach transfers across architectures. Section 4.6 outlines a simple predictive model to map programs to configurations, and Section 4.7 summarizes experimental results. Finally, Section 4.8 summarizes the chapter.

4.1 Introduction

Peyton Jones and Marlow [126], the primary developers of the Glasgow Haskell Compiler, pointed out that inlining is particularly important in functional languages, as it subsumes other optimizations that are performed separately in an imperative setting, such as copy propagation and jump elimination.

In addition, functional programs often contain significantly more functions that need to be considered for inlining, due to the frequent use of anonymous functions (a.k.a., *lambda expressions*) and the cultural encouragement to use function abstractions abundantly. Furthermore, inlining is not restricted to functions but can be performed for every `let`-bound variable. Practical functional programming relies on the ability of optimizing compilers, such as the Glasgow Haskell Compiler (GHC), to aggressively inline function calls and compile away the complex abstractions expressed in user code.

Peyton Jones and Marlow call effective inlining *"particularly crucial in getting good performance"*, state that *"it is our experience that the inliner is a lead player in many [performance] improvements"*, and also *"No other single aspect of the compiler has received so much attention"* [2002]. Similarly, Minsky highlights the significance for OCaml, as *"inlining is about more than just function call overhead. That's because inlining grows the amount of code that the optimizer can look at at a given point, and that makes other optimizations more effective"* [2016]. While OCaml has improved inlining with a new compiler intermediate representation, maybe surprisingly, the approach of GHC to inlining has not changed significantly in the last 20 years.

If this optimization is so critical to functional programming in general, and GHC's performance in particular—why has it not been re-examined, given the massive hardware changes witnessed in the last 20 years? A likely reason is that its inlining decisions are poorly understood, rely on hard-wired constants, and are scattered throughout the compiler. While Peyton Jones and Marlow [126] describe the overall design choices of the GHC inliner and particular implementation challenges, they avoid discussing the crucial numerical parameters that make up the heuristics that eventually decide to inline or not. The heuristics' complex implementation and their reliance on these numbers makes evaluating and modifying GHC's inlining behaviour difficult.

These hand-coded numerical parameters reflect the GHC developers' "best guess" as to what should be inlined, and they are often accompanied by comments expressing the arbitrary nature of the choices made for their values. This highlights them as *magic numbers*, or constant numbers written directly into the source code [109], and makes modification challenging. Furthermore, changes to these parameters—and GHC in general—are still performance tested against the `nofib` benchmark suite described by Partain in 1993. The `nofib` suite itself is falling into obsoleteness, as observed by Marlow already over 15 years ago [2005]. Inlining in GHC is thus a compiler optimization that is thought to be highly significant, yet difficult to modify and evaluate.

Dissatisfaction with the performance of GHC's inliner is highlighted by developers' frequent use of pragmas to manually annotate their code in an attempt to coerce GHC to inline specific functions and improve performance. Our investigations revealed that 1 in 5 of Haskell projects uploaded to Hackage, the Haskell community's central package archive, contain manually inserted "inline" compiler pragmas.

In this chapter, we systematically study inlining in the context of functional languages. We focus specifically on the performance of GHC's inliner, as GHC is one of the most widely used optimizing functional compilers and known to deliver good performance. While our experimental evaluation is specific to GHC, our methodology and findings are of interest to compiler engineers of other functional languages.

4.1.1 Overview of the GHC Inlining Heuristic

The logic for GHC's inlining decisions is scattered throughout the codebase. A search for "CoreUnfold" brings up 30 different files in GHC's compiler directory. We thus present a simplified account of the heuristic, depicted in Figure 4.1.

The `callSiteInline` function (top right of Figure 4.1) is invoked to determine whether to inline or not. Any inlining decision which requires nontrivial consideration is labeled as a `CoreUnfolding` and passed to the function `tryUnfolding` (middle of Figure 4.1), which makes a value judgment based upon the estimated size of the callee, its arguments, how it fits within its context, and other interesting attributes. At a highly simplistic level, it calculates the cost and benefit of inlining: if the cost minus benefit is less than a threshold, then it performs inlining.

The calculation happens in this line

```
small_enough =
    (size - discount) <= ufUseThreshold dflags
```

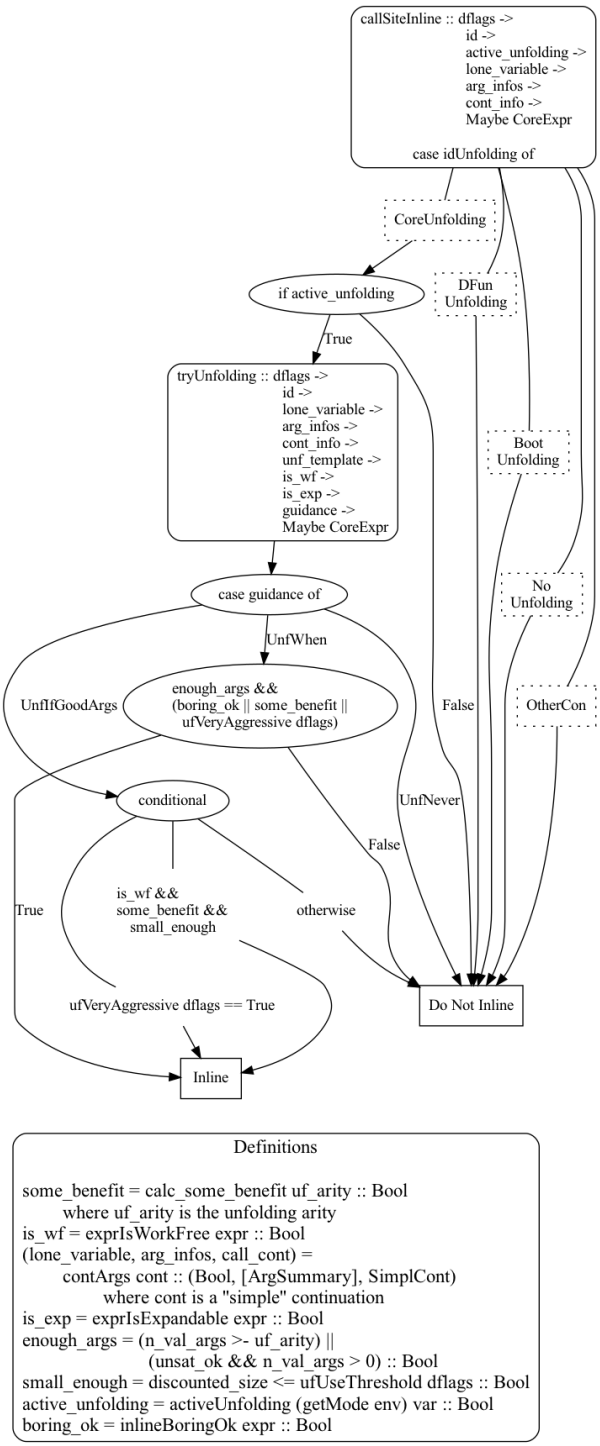


Figure 4.1: Visualization of GHC's Inliner. The function `callSiteInline` is declared in CoreUnfold.hs and is called from Simplify.hs. Rounded boxes indicate functions, ovals indicate conditions, and dotted boxes indicate unfolding IDs.

which determines acceptability for inlining, where `size` is determined by a traversal of the inlinee and `discount` is calculated with consideration to the inlinee's arguments, the continuation, and dynamic flags optionally set upon compilation. The discount represents the potential value gained from inlining, which would offset the cost of inlining large things. This computation happens when the Simplifier—a module where GHC iteratively applies optimizations to the Core intermediate representation (Core IR) code—calls `tryUnfolding` on a `CoreUnfolding`.

Each inlining decision additionally depends upon considerations including but not limited to: the type of the expression, its arity, its number and characterization of arguments, the phase of compilation, and a number of calculated discounts and thresholds written directly into GHC simply as best-judgment constants.

4.1.2 Magic Numbers in the Inliner

The calculations for both `size` and `discount` rely upon several magic numbers written directly into the inliner. An example of the use of these numbers occurs in the first few lines of the function `computeDiscount`, shown in Figure 4.2, which computes a discount value for all functions being considered for inlining. In `computeDiscount`, the number 10 refers to a discount given for the function itself.

```
computeDiscount :: [Int] -> Int -> [ArgSummary]
                         -> CallCtxt
                         -> Int
computeDiscount arg_discounts
                res_discount
                arg_infos
                cont_info
  = 10 -- Discount of 10 because the result
       -- replaces the call
       -- so we count 10 for the function itself
```

Figure 4.2: First part of the `computeDiscount` function, with the magic number 10, in CoreUnfold.hs.

In this example, making the number 10 larger would give the inlinable item a larger discount to offset its size, increasing its likelihood to be inlined. Such a modification would make *all* functions more likely to be inlined because they would start with a higher base discount, before the addition of any discounts based upon their arguments. These would be calculated in the lines immediately following in `computeDiscount`, as shown in Figure 4.3.

```
  + 10 * length actual_arg_discounts
  + round (ufKeenessFactor dflags *
        -- Discount of 10 for each arg supplied,
        -- because the result replaces the call
      fromIntegral (total_arg_discount+res_discount'))
```

Figure 4.3: Second part of the `computeDiscount` function, with the magic number 10, in CoreUnfold.hs.

Additionally, the term `res_discount'`, short for "result discount", adds a discount when an efficiency is expected to be gained through inlining—for example, through case reductions. Its numerical value is computed by considering a simplified version of the context, represented by the data type `CallCtxt`. The original code of Figure 4.4 shows how some of these possible `CallCtxt` values are assigned to the magic number 40 to return as a result for `res_discount'` in one single line of code, along with the comments right after it which debate its accuracy.

```
_            -> 40 `min` res_discount
    -- ToDo: this 40 `min` res_discount doesn't
    -- seem right
    --   for DiscArgCtxt it shouldn't matter because
    --     the function will get the arg discount
    --     for any non-triv arg
    --   for RuleArgCtxt we do want to be keener to
    --     inline; but not only constructor results
    --   for RhsCtxt I suppose that exposing a data
    --     con is good in general
    --   And 40 seems very arbitrary
```

Figure 4.4: GHC 8.10.3: CoreUnfold.hs, line 1640. The top line of code calculates the value for `res_discount'` seen in Figure 4.3. The developer's comments highlight some of the arbitrary decisions made.

Magic numbers such as these are scattered throughout the entire inliner, and its decisions are fundamentally dependent upon them. We set out in this chapter to study the impact of these magical numbers systematically.

4.2 Approach

For this study, we wanted to answer the question: *Could a modification to the inliner's thresholds yield a performance improvement across Haskell code execution time?* If the answer to that question is *yes*, then we face two additional questions: If so, how much improvement might we expect to see by modifying GHC's inlining thresholds? If not, how should we then modify GHC to attain an optimal improvement?

It is necessary to answer these questions before redesigning the inliner, given the complexity of the system. Thus, we constructed a set of benchmarks with the intention of revealing weaknesses in GHC's inlining decision process. We then modified GHC 8.10.3 such that we could change its inliner's magic number values through dynamic flags to see how much we could affect the benchmarks' execution times through the inliner alone.

4.2.1 Optimization Space Exploration

Because parameterizing all of the inliner's thresholds would have been intractable, we focused on 10 hand-coded magic-number constants to expose as dynamic flags, which could then be passed into GHC when compiling an application. Additionally, in our optimization space, we included two of GHC's built-in dynamic flags. Combined, this totals 12 parameters.

To approximately quantify the type of inlining decisions being performed, we added hooks to GHC 8.10.3 and compiled it against the Cabal library, where Cabal is the canonical system for building and installing Haskell packages. During compilation, GHC performed 8,708,142 nontrivial inlining decisions, where "nontrivial" means any inlining for which it is not obvious that it should definitely be inlined. Among these nontrivial inlinings, 81.8% were designated as `UnfIfGoodArgs`—which means their unfolding would be large enough to require consideration, but not so large to immediately disqualify it from inlining. Before deciding whether to inline, GHC gives these potential inlinings a reduction in their calculated sizes via a discount calculation:

$$\text{discounted\_size} = \text{size} - \text{discount} .$$

We therefore decided to create parameters from magic numbers involved in the calculation of `size` and `discount`.

4.2.2 Characterization of the Parameters

Each parameter was selected because it had a direct impact on GHC's inlining decisions and would likely produce an observable effect on runtime performance. Table 4.1 describes each parameter and gives their names and original values.

Flag	Description	Original Value
nontrivarg-disc	Discount for an argument labeled “NonTrivial”.	10
funcitself-disc	Constant discount value added to every function inlined.	10
actarg-disc	Discount for each argument.	10
discargctxt-disc	Context is the argument of a function with non-zero argument discount.	40
ruleargctxt-disc	Context is the argument of a function with rules.	40
rhsctxt-disc	The context is the right-hand side of a let.	40
arbctxt-disc	The wild card remaining to catch any other type of context and calculate its discount.	40
cosbase	Base size value of a class op.	20
cosargs	Size metric added for each argument of a class op.	10
bigalt	Size component of the biggest alternative when scrutinizing a case expression argument.	20
funfolding-fun-discount	Adjust the eagerness of GHC to inline functions.	60
funfolding-dict-discount	Adjust the eagerness of GHC to inline dictionaries.	30

Table 4.1: Inlining parameter dynamic flags, their descriptions, and original values.

Three parameters, *cosbase*, *cosargs*, and *bigalt*, calculate various components of an inlinee's size. The remaining 9 help calculate its discount, or the numerical value estimated to offset the cost of inlining. We also included the built-in GHC dynamic flags *-funfolding-fun-discount* and *-funfolding-dict-discount*, as they both pertained specifically to inlining.

4.2.3 Benchmark Construction

To experimentally evaluate the performance of GHC's inliner, we needed a benchmark suite that would allow us to analyze the performance impact of different inlining decisions.

The `nofib` benchmark suite was originally constructed to be a substantial, diverse, relevant set of programs in 1993; but now, most of its programs run for a fraction of a second, as pointed out by Marlow [105]. Unfortunately, despite its age, nofib has yet to be replaced or upgraded. As an alternative, we wanted to allow developers to experiment and evaluate on interchangeable, testable, real-world packages from Hackage so that the resultant benchmarks would be heterogeneous and relevant to real-world needs. We based that assumption on previous work to construct a benchmark suite for JavaScript, as described by Richards et al. [133].

We therefore constructed a tool in Python to select Hackage packages specifically to suit our benchmarking goal: to identify room for execution time improvement as it pertains to inlining. These programs needed to run for an adequate amount of time, perform a variety of different tasks, and have consistent execution times such that the same inlining decisions would reproduce the same results.

The benchmark framework itself therefore had the capability to:

1. Download packages by url,
2. Build the packages,
3. Control for noise introduced by generation of random tests, and
4. Record timings of tests or benchmarks present in the packages.

Section 4.2.4 gives more information about controlling for noise introduced by random test generation.

4.2.4 Benchmark Selection

Stackage is a distribution of a subset of Hackage, where packages within the same snapshot will build together and pass all of their tests. For our benchmarks, we selected packages contained within a single Stackage snapshot.[1] In this Stackage Nightly build, 854 of 2218 packages (about 39%) used QuickCheck, a tool which generates random tests developed by Claessen and Hughes [34]. Initially, these randomly generated tests were a significant source of unwanted noise. To address this, we set QuickCheck's random seed to one constant and made its test times consistent. We then enabled our scripts to automatically patch all selected packages' dependencies with our modified QuickCheck.

In our Stackage snapshot, 421 of the 2218 packages contained `INLINE` pragmas—or about 19%. We hypothesized these packages may provide code where developers had identified a good set of problems upon which to evaluate inlining.

Influenced by our observation of pragmas, we identified 236 packages with INLINE pragmas in their "src" folders that could be run with `cabal new-test`. From those packages, we sub-selected 10 which each ran over 4 seconds, decreasing the likelihood that any speedup percentages observed would fall outside the range of noise. Table 4.2 characterizes the selected 10 packages.

---

1. stackage−nightly−2020−01−31

Package	Version	SLOC	Description	Default Sec.	INLINE Pragmas
hw-rankselect	0.13.3.1	1387	Efficient rank and select operations on large bit-vectors	8.18	88
ListLike	4.6.3	3402	The ListLike package provides typeclasses and instances to allow polymorphism over many common datatypes.	23.04	2
loop	0.3.0	155	Fast loops (for when GHC can't optimize forM_)	19.94	8
metrics	0.4.1.1	1819	High-performance application metric tracking	58.48	9
midi	0.2.2.2	5094	Handling of MIDI messages and files	19.18	2
monoid-subclasses	1.0.1	4900	Subclasses of Monoid	35.12	334
nonempty-containers	0.3.3.0	10055	Non-empty variants of containers data types	4.38	520
poly	0.3.3.0	2040	Haskell library for univariate and multivariate polynomials, backed by Vector.	94.56	57
reinterpret-cast-0.1.0	0.1.0	122	Memory reinterpretation casts for Float/Double and Word32/Word64	26.86	3
set-cover	0.1	2781	Solve exact set cover problems like Sudoku, 8 Queens, Soma Cube, Tetris Cube	15.55	16

Table 4.2: Selected Stackage packages and their information. Source lines of code (SLOC) are estimates. Descriptions were taken from the packages' Hackage profiles.

4.2.5 Pragma Example

User-inserted compiler pragmas may hint that a compiler's optimization decisions could be improved. This snippet from poly contains the INLINE pragma `{-# INLINE integral #-}`:

```
-- | Compute an indefinite integral of a polynomial,
-- setting constant term to zero.
--
-- >>> integral (3 * X^2 + 3) :: UPoly Double
-- 1.0 * X^3 + 3.0 * X
integral :: (Eq a, Fractional a, Vector v (Word, a)) =>
            Poly v a -> Poly v a
integral (Poly xs) = Poly
  $ map (\(p,c) -> (p+1, c/(fromIntegral p + 1))) xs
{-# INLINE integral #-}
```

Here, the function `integral` is overloaded. Without inlining it, `integral` would get passed a dictionary of functions for the possible types of '`a`'.

When `integral` is inlined, GHC may see that '`a`' has a specific type—for example, float—and then specialize for it. In this way, we can sometimes substitute the retrieval and application of unknown higher-order functions with single machine instructions by telling GHC to inline with pragmas. This makes the resultant code much faster.

4.3 Experimental Setup

To analyze, explore, and improve the performance of the GHC inliner, we perform an in-depth experimental evaluation on our benchmarks. In all experiments, programs are executed 10 times and average time is reported. The default baseline is the execution time of a package compiled with unmodified GHC 8.10.3 and `INLINE` pragmas disabled. We refer to such execution times as *without pragmas*. If `INLINE` pragmas are enabled, this is referred to as *with pragmas*.

We wanted to explore both parameter values which were likely to yield good performance and also values from a larger range; therefore, we sampled from both a normal distribution and a uniform distribution. Sampling from a normal distribution stays near GHC's original values at the mean and assumes that they are reasonable values. The normal distribution therefore takes μ as the original flag's value and $\sigma = 0.4$. If the generated number was negative, number generation recurred until sampling produced a positive value. We ran 140 configurations randomized in this manner: 70 on the packages with pragmas and 70 without.

We ran additional configurations from a uniform distribution with a lower bound of 0 and an upper bound of $2 * \mathtt{N}$, where N was the default value. For this experiment, we collected 250 configurations without pragmas and 250 with pragmas. The final result contained 640 randomly sampled data points, 320 without pragmas and 320 with pragmas. When we evaluate the performance impact of searching for good configurations, we refer to this as *search*. We ran all benchmarks in isolation, sequentially, on a dedicated server (AMD EPYC 7720P CPU, 2.0 GHz and $256\,\mathrm{GB}$ RAM).

4.4 Experimental Results

We first examine the impact of pragmas and a per-program parameter configuration search on the benchmarks. Then we analyze both the best-performing parameter values, as determined by execution time, and their impact on inlining decisions. Next, we evaluate the performance of the best single fixed-parameter configuration for all programs. We then evaluate and analyze a simple cluster-based model to predict good configurations for an unseen program. Finally, we examine to what extent inlining is architecture-dependent.

4.4.1 Performance Improvement

Figure 4.5 shows the best performance improvements found when adding pragmas and configuration search on a per-program basis. All best configurations reported in this figure came from the uniform distribution. The respective geometric means across all programs are summarised in Figure 4.6.

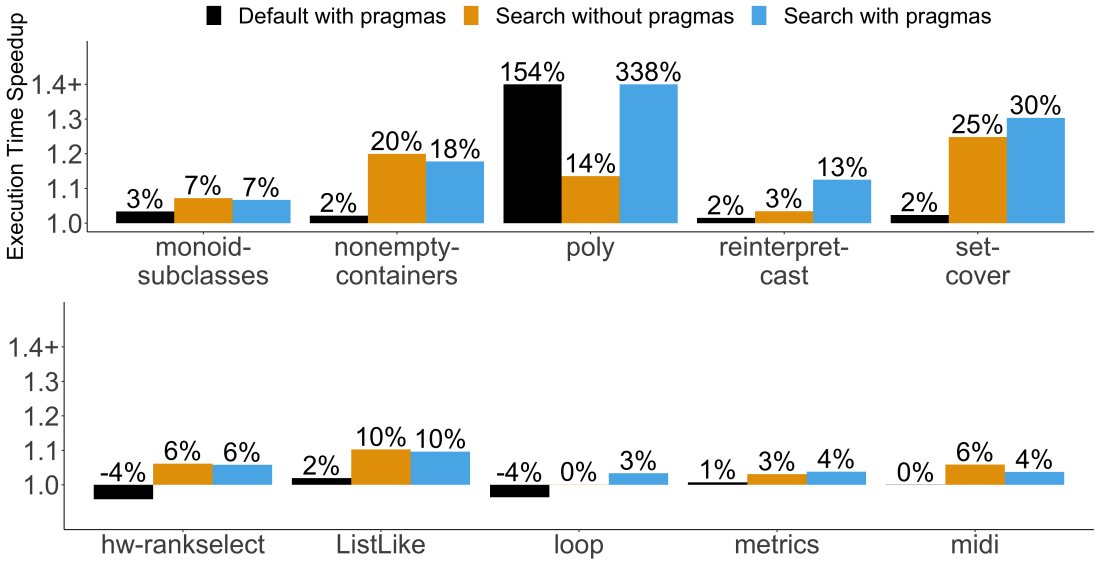


Figure 4.5: Best-case speedups for each package, grouped by experiment. Speedups are reported as run time ratio along the x-axis and labelled with speedup percentages at the top of bars. The baseline is default GHC without pragmas in package code.

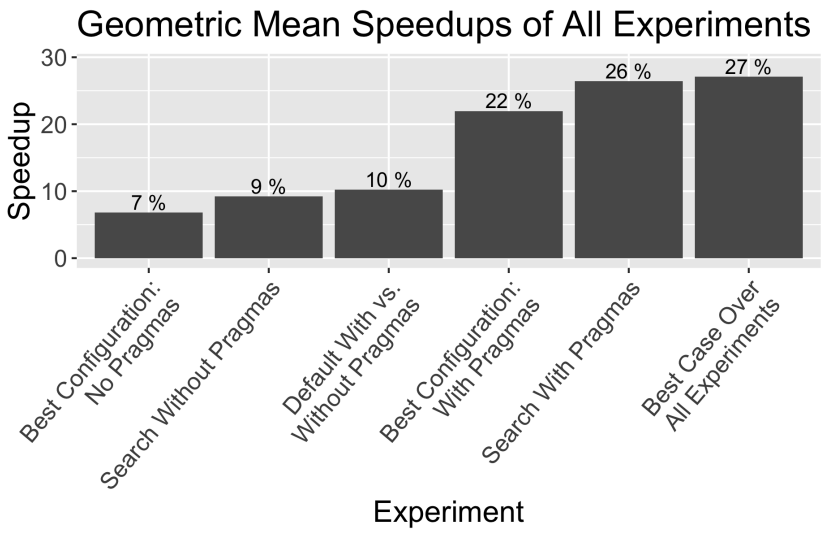


Figure 4.6: Geometric mean speedups for all experiments. Baseline: Default, without pragmas. "Search With Pragmas" and "Search Without Pragmas" show geometric mean speedups of the averaged best configurations for each package. "Best Configuration" experiments represent the single best configuration applied to all 10 packages.

Pragmas. "Default with pragmas" in Figure 4.5 shows the performance difference when `INLINE` pragmas are included in the code versus the baseline (code with pragmas removed). The geometric mean speedup for this experiment was 10%, as shown in Figure 4.6. However, most of this improvement came from one package, poly, which had a 154% speedup. Although six of the other packages had a speedup above zero, only one had a speedup above 3%. In fact, two packages had negative speedups at -4%, where the pragmas actually had a detrimental effect on execution time. While pragmas can improve the performance of Haskell code, their use shows a variance of results. This may have multiple explanations, three of which are 1) developers may insert pragmas where GHC would normally inline anyway, 2) the effectiveness of the pragmas has changed over time or architectures, or 3) the effectiveness of the pragmas is not verified by or reflected in test cases.

Search. If we search and evaluate configurations in our space on each program without pragmas enabled and report the best value, we get the results labeled "Search without pragmas" in Figure 4.5. If we repeat this search experiment with pragmas enabled, we get the results in Figure 4.5 labeled "Search with pragmas".

As the results in Figure 4.6 show, searching for the best configuration on a per-program basis significantly improves performance. Although searching with pragmas gives, on average, better performance than searching without pragmas, Figure 4.5 shows there are 5 programs where searching without pragmas gives a better individual speedup than the other two experiments: monoid-subclasses, nonempty-containers, hw-rankselect, ListLike, and midi. If we calculate the geometric mean of the best speedups across all experiments, we achieve the maximum possible speedup of 27% shown in Figure 4.6. Finally, independent of pragmas, searching always delivers a performance improvement. Performance improves without pragmas by 9% and actually has a greater *additional* impact of 16% (26% vs 10%) on packages with pragmas.

Speedup distribution. Figure 4.7 shows the histograms of speedups achieved with and without pragmas. While the majority of results are clustered around 1 (where 1 indicates no change in execution time), there are some significant positive and negative outliers. For the configurations with pragmas, a speedup over default was observed 64% of the time; and for the configurations with pragmas removed, a speedup was observed 50% of the time. This reflects the earlier observation that inline pragmas, on average, improve performance.

In Figure 4.8, we examine the best speedup achieved over time as we search the configuration space. The x-axis is the number of configurations evaluated, while the y-axis reports the best speedup achieved so far. There are two lines representing presence of pragmas, solid with pragmas and dotted without pragmas. The baseline is default GHC without pragmas. While most speedup is achieved early in the search, many configurations are needed to find the best performance.

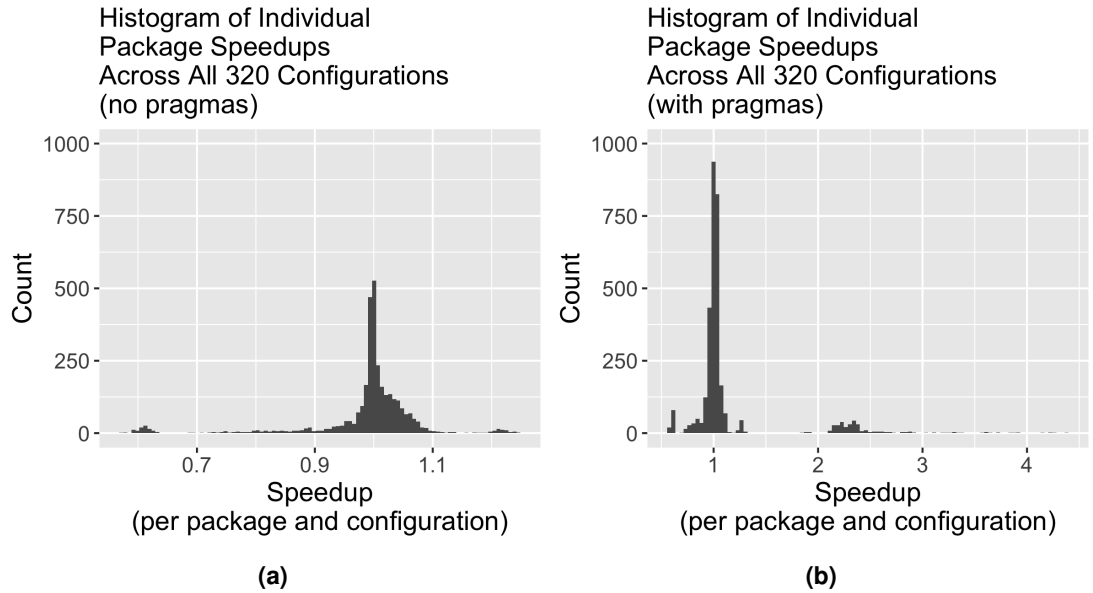


Figure 4.7: Histogram of individual package speedups from search across 320 configurations *without* (a) and *with* (b) package INLINE pragmas.

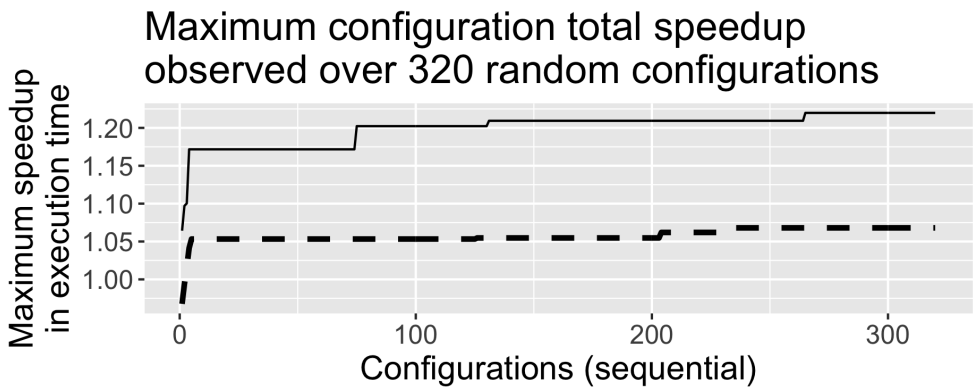


Figure 4.8: Maximum total speedup of the single best configuration observed over time, search without pragmas (dotted line) and with pragmas (solid).

4.5 Analysis

In this section, we examine how the best values of parameters vary across programs.

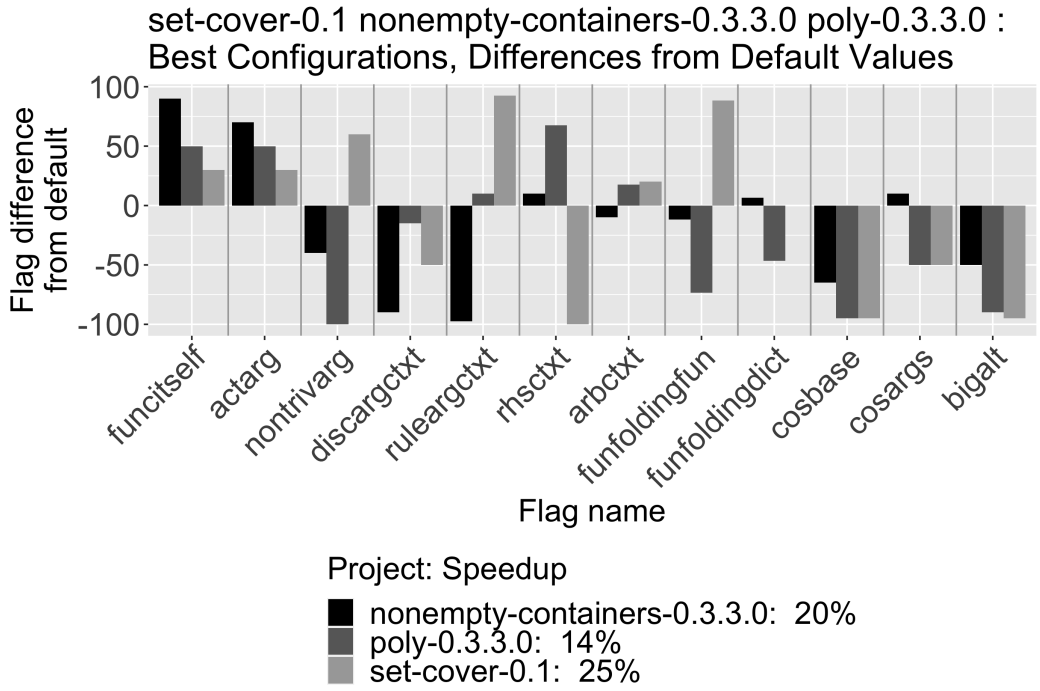
Parameters

Distribution. In Figure 4.9, we took the single best configuration for each package and plotted its values for each parameter. Optimal values for each flag, and for each package, are spread across the range of the random distributions from which they are sampled—with the exception of *funfoldingfun*, which almost entirely prefers a value above 50 (minus one data point).

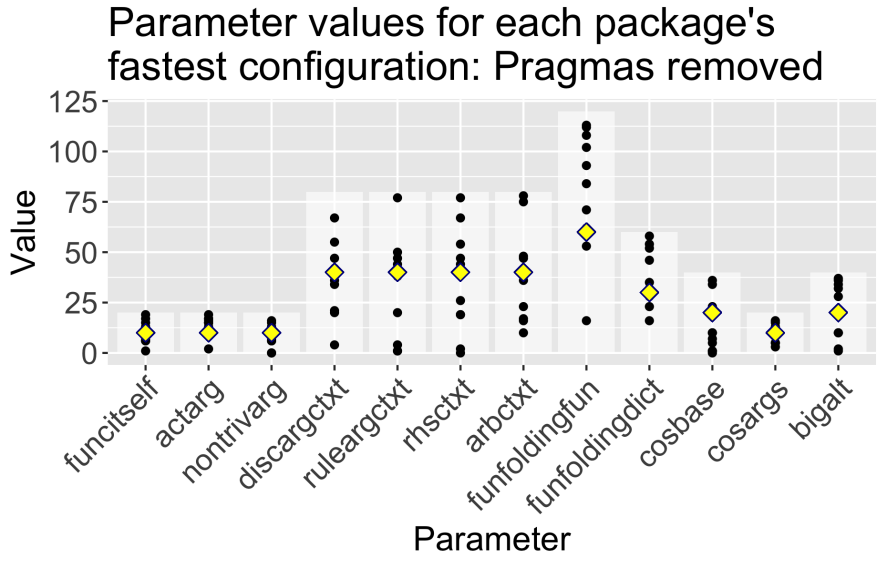
Looking at a wider set of the most performant configurations, in Figure 4.10 we filtered the data to include configurations within 1% of the optimal value for each project. We additionally excluded configurations with less than a 3% speedup, which removed all configurations for the packages loop and metrics. In the remaining data, the flag *actarg* seems to clearly prefer a value higher than its default (except for 3 configurations in which its value is 0); the flag *cosbase* mostly prefers a value lower than its default; but overall, all of the flags contain values from the lower and upper bounds of their random distributions.

Variation across most-improved programs. If we examine the three best-performing programs in more detail, Figure 4.9a shows the percent difference from the default values of the best flag configurations for these three packages (without pragmas). The three packages in question, poly, nonempty-containers, and set-cover, have respective speedups of 14%, 20%, and 25%. For almost any flag, the three packages strongly differ—with the flag *arbctxt* having the least disagreement at +20%,-10%, and +17.5%. Where Figure 4.9b suggests all packages' best-case configurations differ widely on their ideal values, Figure 4.9a confirms that this disagreement holds even among the three packages with the highest observed speedups.

Reflecting on intuition. We case matched the context parameters *discargctxt*, *ruleargctxt*, *rhsctxt*, and *arbctxt* to collect values to address the comments in Figure 4.4. Recall the developer deliberated over what the value of the magic number for the call context should be and speculated that the value for `DiscArgCtxt` should not matter, the value for `RuleArgCtxt` should perhaps be higher ("keener to inline"), the value for `RhsCtxt` should probably be high to expose the inlining, and 40 seems rather arbitrary for all of them. The data disagrees with the comments on the two points of: *ruleargctxt* seems to slightly prefer being at or *below* 40, and so does *rhsctxt*. For all four contexts, the range of their values in optimal configurations varies widely.



(a)



(b)

Figure 4.9: Top (a): Difference from default for each flag, by top configuration for 3 most improved programs, without pragmas. **Bottom (b)**: Values for each single best configuration for each package. Yellow diamonds are default values. White bars indicate sampling boundaries.

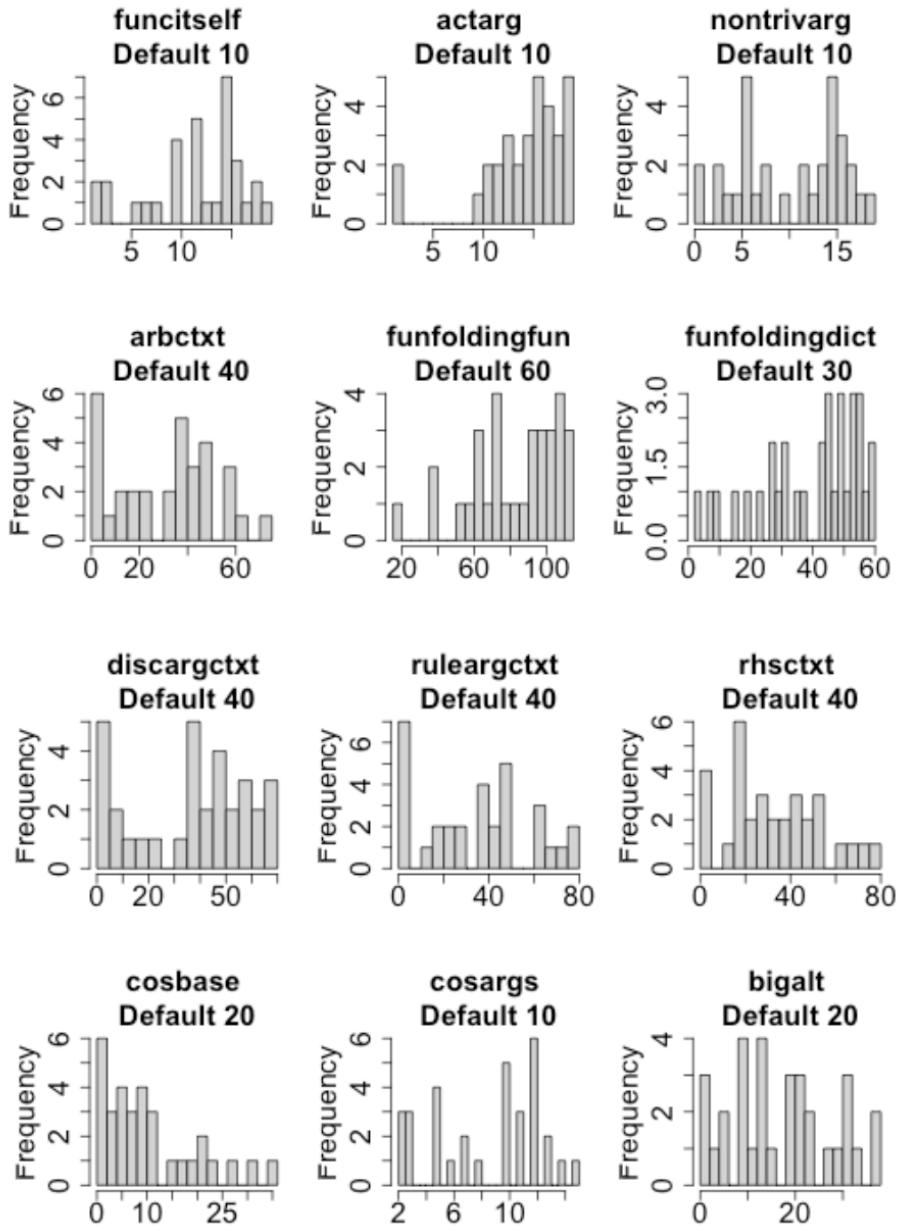


Figure 4.10: Histograms of parameter values for configurations within 1% of the optimal speedup for each package, across all samplings.

Inline Decisions

Table 4.3 shows a comparison of GHC's default inlining behavior to that of the best configuration for each package. To summarize, every best configuration considers more total items for inlining than default GHC, with an averaged 48% increase in number of combined yes and no decisions. Nine of the ten packages decide to inline more total items than default GHC,

Package	Total Decisions		% Inlined		Avg. Size Inlined	
	Default	Best	Def.	Best	Def.	Best
hw-rankselect-0.13.3.1	1,166,781	1,515,164	6.0%	5.7%	14	18
ListLike-4.6.3	910,967	999,237	7.8%	7.4%	15	19
loop-0.3.0	363,940	384,689	6.0%	6.0%	21	25
metrics-0.4.1.1	355,448	363,286	6.5%	6.4%	19	20
midi-0.2.2.2	713,076	1,140,791	6.9%	4.8%	26	31
monoid-subclasses-1.0.1	954,442	1,668,907	5.1%	3.6%	19	29
nonempty-containers-0.3.3.0	920,901	1,769,534	5.3%	2.8%	26	31
poly-0.3.3.0	1,405,465	2,635,700	7.1%	3.1%	17	32
reinterpret-cast-0.1.0	361,600	383,446	6.0%	5.8%	20	22
set-cover-0.1	414,699	877,758	6.0%	3.1%	19	27

Table 4.3: Inlining decisions per package, default vs best magic number configuration.

with the exception of `poly-0.3.3.0`. However, all packages inline a smaller percentage of the decisions, relative to total decisions, than default. Additionally, all packages' best configurations decided to inline larger items, with an averaged inlining size of 25.4 versus the default's averaged size of 19.6.

Characterizing Good Inline Decisions

To better understand the inlining behavior of the default versus best configurations, we collected information vectors for the inlined code in both cases containing information about the compiler intermediate representation (Core IR) at each inlined site, along with the *yes* or *no* decision. Summaries of the derived IR vectors are shown in Table 5.5.

Collection of the Core IR features. To collect features of the inlining decisions, we parsed both the body of the expression being inlined and the calling context, for every nontrivial inlining decision in the benchmarks. Additionally, we included the variables `lone_var` (LVAR); `size` (SIZE); `is_wf` (WF); the type of unfolding (FOLDTY); and the number of arguments to the expression (ARGS). In each decision, we tallied the number of occurrences of expression and continuation Core IR features in a bag-of-words manner, then appended the values for LVAR, WF, FOLDTY, SIZE, and the *yes* or *no* inlining decision.

Abbr.	Value	Description
Continuation Features		
CSt	Int	Stop[e] = e
CCI	Int	CastIt co K)[e] = K[e ‘cast‘ co]
CAV	Int	(ApplyToVal arg K)[e] = K[e arg]
CAT	Int	(ApplyToTy ty K)[e] = K[e ty]
CSe	Int	(Select alts K)[e] = K[case e of alts]
CSB	Int	(StrictBind x xs b K)[e] = let x = e in K[\xs.b] or equivalently = K[(\x xs.b) e]
CSA	Int	StrictArg (f e1 ..en) K)[e] = K[f e1 .. en e]
Argument Features		
AINT	Int	The argument has structure.
AIVA	Int	The argument is a constructor application, partial application, or constructor-like.
AITA	Int	The argument is not interesting, i.e., deserves no unfolding discount.
ARGS	Int	The number of arguments
Expression Features		
LVAR	Int	Indicates if the expression is a lone variable.
IFV	Int	Number of vars that don’t occur in a coercion.
IFJI	Int	Number of join variables
IFL	Int	Number of literals
IFCase	Int	Number of case expressions
IFR	Int	Number of recursive lets
IFLNR	Int	Number of non-recursive lets
IFCast	Int	Number of cast expressions
IFLam	Int	Number of lambda abstractions
IFApp	Int	Number of applications
FOLDTY	{ 0, 1 }	Type of unfolding: UnfWhen or UnfIfGoodArgs
SIZE	Int	The size of the expression
WF	{ 0, 1 }	GHC estimate if expr. will not duplicate work

Table 4.4: Collected IR data: their abbreviations, possible values, and descriptions.

The **Continuation Features** in Table 4.4 are constructors of the data type `SimplCont` describing a strict context that does not bind any variables. It represents the rest of the expression, above the point of interest, and allows GHC’s Simplifier to traverse it like a zipper. The inlinee’s expression is represented by the features in **Expression Features**, where the values `Var`, `Lit`, `Case`, `Cast`, `Lam`, and `App` are constructors of the recursive data type `CoreExpr`. The features IFR and IFLNR indicate recursive `let` and non-recursive `let`, respectively, which are pattern matched against compositions of the values `Let`, `Rec`, and `NonRec`.

Analysis. We averaged the collected feature vectors across all call sites and programs and compared the difference between the default and best configurations, as shown in Figure 4.11. For some features there is little difference, such as `Case` expressions (IFCase), `Stop` values (CSt), `join` variables (IFJI), and work-free expressions (WF). For other features, there is significant difference—such as far fewer inlinings for `CastIt` (CCI) and `StrictBind` (CSB), and far more inlinings for non-recursive `lets` (IFNLR) and recursive `lets` (IFR). Additionally, the sizes of the best-case inlinings are 31% larger than GHC's default inlinings. As an explanation, non-recursive `let` is an example of an inlining decision associated with anonymous functions particular to functional languages. In summary, then, these features suggest that more inlinings should be performed over anonymous functions, and larger things should be inlined.

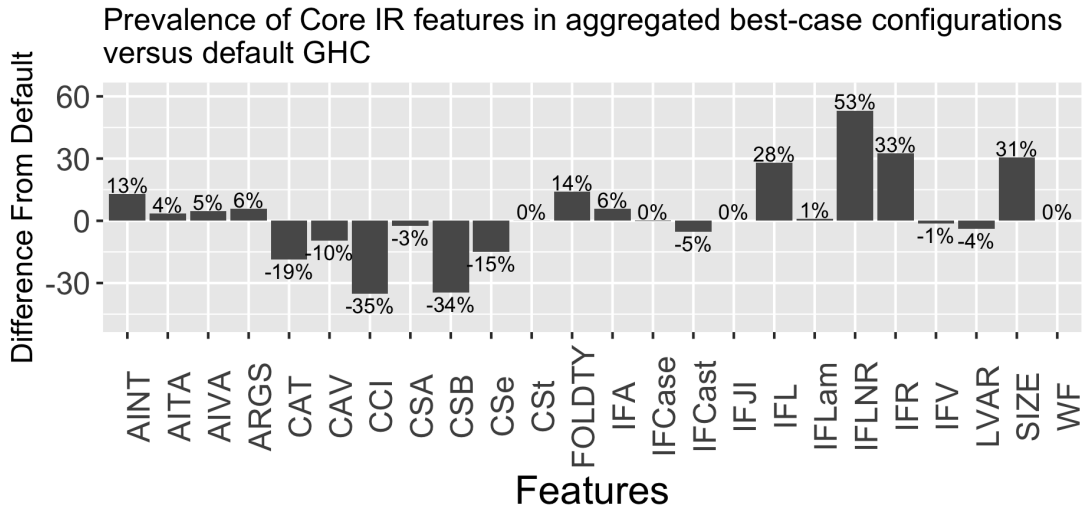


Figure 4.11: Difference of inlining features, best-case configurations compared against default GHC.

4.5.1 The Single Best Configurations

While searching for good configurations yields better performance, it is interesting to ask if a single fixed configuration can perform better than the default across all programs. If so, this could replace the existing hard-wired numeric values. Figure 4.12 shows the speedups achieved when applying the two best single configurations recorded across all of the packages: the single best configuration with pragmas and the single best configuration without pragmas, respectively. Table 4.5 shows these two configurations' parameter values. On average, a mean speedup of 7% is achieved with pragmas disabled, and 22% with pragmas enabled. Thus, when searching for ideal configurations is too expensive, then using a new single best configuration gives already significant improvement.

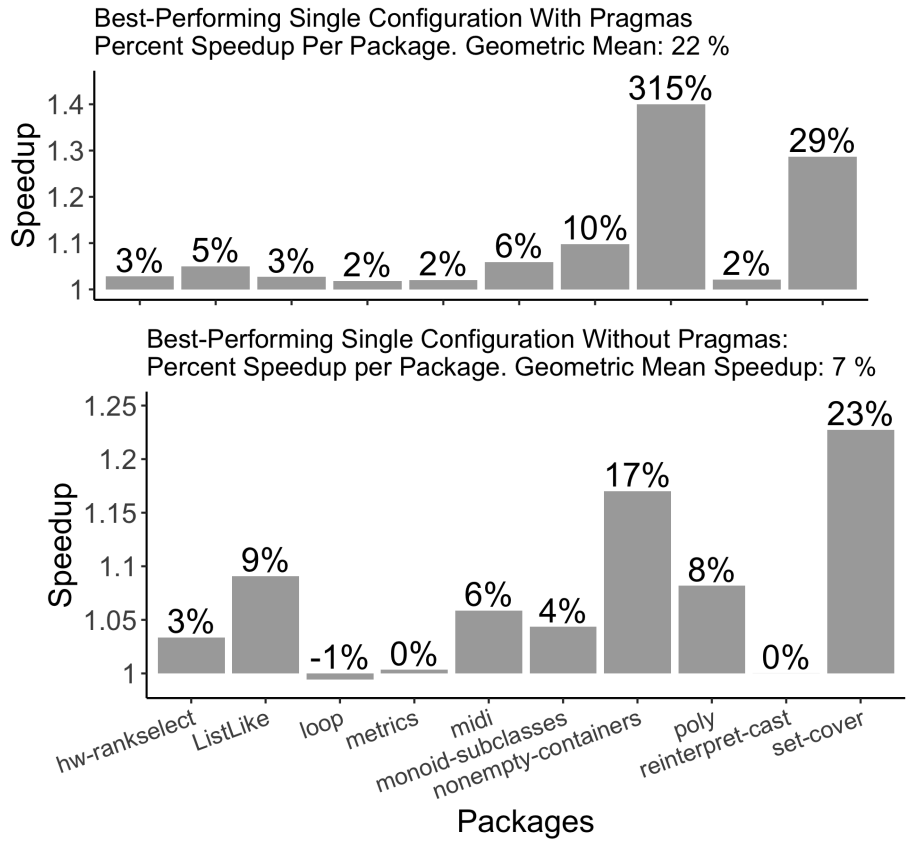


Figure 4.12: Best-performing single configurations. Top: with pragmas. Bottom: without pragmas. All speedups relative to baseline of unmodified GHC times without pragmas.

4.5.2 Cross-Architecture Transference

To investigate whether inlining behavior is independent of platform, we evaluated the best performing configurations on a new machine. The second machine was an Intel Xeon CPU E3-1270 v6 with 4 cores running at $3.8\,\mathrm{GHz}$ with $62\,\mathrm{GB}$ RAM on Debian GNU/Linux 10.

The results are shown in Figure 4.13. On average, we are able to obtain a 6% and 21% improvement without and with pragmas, respectively. These numbers are promisingly close to results found on the original machine, where 9% and 26% were respectively found. These encouraging results show that improvements to the inliner may port across machines; however, further work will be needed to confirm this.

Parameter	229 (Without Pragmas)	265 (With Pragmas)	GHC
nontrivarg-disc	15	11	10
funcitself-disc	6	1	10
actarg-disc	17	17	10
discargctxt-disc	55	40	40
ruleargctxt-disc	38	60	40
rhsctxt-disc	54	19	40
arbctxt-disc	23	34	40
cosbase	0	22	20
cosargs	15	2	10
bigalt	37	38	20
funfolding-fun[1]	71	115	60
funfolding-dict[2]	58	49	30

Table 4.5: Parameter values for configuration 229 (single best without pragmas) and 265 (best with pragmas). Default GHC values in rightmost column. 1) funfolding-fun-discount. 2) funfolding-dict-discount.

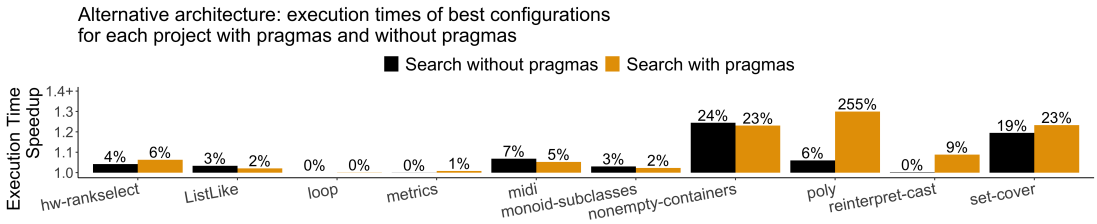


Figure 4.13: Execution times for best-case configurations for each project on alternative architecture. Geometric mean speedup of configurations without pragmas: 6%. Geometric mean speedup of configurations with pragmas: 21%.

4.6 A Simple Machine Learning Predictive Model

We saw that when searching for the best configuration per program, we achieve an average speedup of 26%; however, only 22% is achievable with a single, fixed, best-on-average configuration. Next, we investigate whether a simple machine learning approach can improve performance without the need to search many configurations to find an optimum.

We use a simple top-down dynamic programming algorithm for clustering: start with the single configuration that has the most programs at their optimal performance as the initial cluster, then place the poorest performing program into a new cluster within its own best configuration along with any other program that performs better in that configuration. Recur on this step until just before the number of desired clusters is exceeded.

With a new unseen program, we must determine which cluster it belongs to based on similarity of features. Although static or dynamic program features could be used, we instead use the speedups of the program on a set of selected configurations to determine which cluster it belongs in. Depending on the speedups the program exhibits against each configuration, we allocate it to the cluster of the fastest speedup and assign it that configuration of magic numbers.

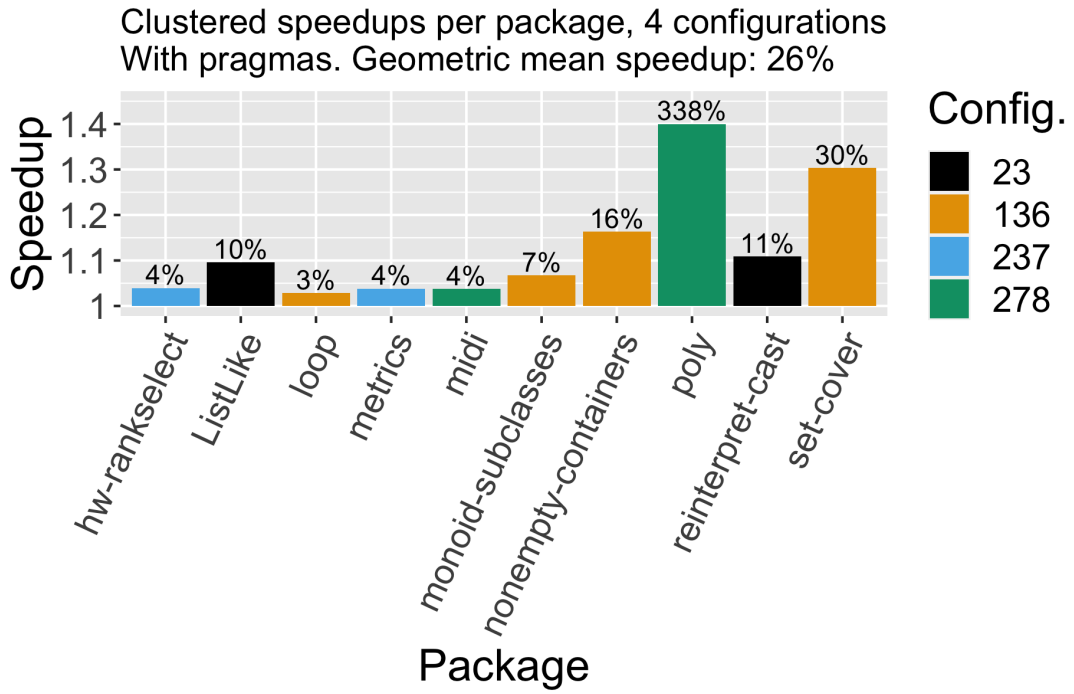


Figure 4.14: Performance of model. Speedups for each package, using a 4-cluster based predictive model with pragmas.

Analysis. To analyse the results, we focus on a single clustering, where we trained the model without the first benchmark and then assigned it to the best cluster. This is presented in Figure 4.14, showing an average speedup of 26% across the benchmark suite.

Table 4.6 shows the values of the parameters in each cluster configuration. It can be highlighted that some parameters are similar across clusters, but seldom in complete agreement. The `cosargs` parameter has the closest values to each other at 13, 12, 15, and 11—also near the default 10. The `funfolding-fun-discount` parameter is consistently higher than default GHC across all four configurations. Three clusters (136, 237, and 278) have very close values of `bigalt` near 38, but cluster 23 prefers this value at 9; and similarly, three clusters prefer a `discargctxt-disc` value in the twenties, yet cluster 136 prefers a value of 60. This further supports the possibility that no single configuration will ever approach the optimal improvement observed so far for each type of program.

Parameter	136	23	237	278	GHC
nontrivarg-disc	6	5	6	14	10
funcitself-disc	11	13	7	13	10
actarg-disc	18	16	14	11	10
discargctxt-disc	60	28	29	22	40
ruleargctxt-disc	0	32	13	24	40
rhsctxt-disc	56	17	39	54	40
arbctxt-disc	64	69	32	35	40
cosbase	1	17	38	7	20
cosargs	13	12	15	11	10
bigalt	37	9	38	38	20
funfolding-fun[1]	71	101	77	108	60
funfolding-dict[2]	43	21	17	57	30

Table 4.6: Parameter values for configurations 136, 23, 237, and 278 in Figure 4.14. Default GHC values shown in rightmost column. 1) funfolding-fun-discount. 2) funfolding-dict-discount.

4.7 Results Summary

Our experiments demonstrated that a change of the GHC inliner could yield a significant performance improvement, demonstrably up to 27%, which we attained by modifying the inliner's magic numbers. However, much of that improvement is uncovered with the help of `INLINE` pragmas already written into the cody by package developers.

A single new magic number configuration, with help from pragmas, can get us to a 22% improvement over default GHC—and 26% when packages are clustered into 4 different configurations of magic numbers. Without pragmas, a single best configuration can give us a 7% improvement, and we can get a 9% speedup with a 4-configuration clustering. These configuration changes have been shown to improve performance across architectures as well.

To achieve the maximum speedup observed with the help of pragmas, however, the entirety of GHC's inliner should be rethought. The data suggests that no single set of magic numbers will optimize all different types of programs. A newer inlining heuristic should also give more consideration to anonymous functions, and it should inline larger things. The complexity of this problem indicates that machine learning would be a good alternative to further hand tuning.

4.8 Summary

This chapter introduced a benchmark framework which constructed benchmarks by running executable code from real-world Haskell packages selected from Hackage. Controlling for nondeterminism in the packages' random test generation allowed us to turn tests with varying execution times into executables with consistent execution times, upon which we could measure changes to GHC's inlining heuristic.

We then presented an empirical exploration of the GHC inlining heuristic's decision space through modification of hand-coded constants in GHC's inliner, which are also known as magic numbers. Although a relatively small number of magic numbers are parameterized, randomized, and explored to effect inlining decisions, the observed performance changes are considerable across a selection of 10 benchmarks from Hackage, suggesting ample room for improvement in GHC's inlining decision making process.

We reiterate, however, that simply changing magic numbers does not achieve the full observed speedup when single configurations are applied across all the packages. These results motivate a rethinking of GHC's inliner altogether, which we attempt in Chapter 5 from inside GHC's Simplifier; however, we will also see that the experimental exploration of inlining decisions in Chapter 5 guides us to a more effective control-flow directed approach discussed in Chapter 6 at the source-code level, which effects the desired significant speedups.

Chapter 5

Investigatory Work Towards Improving the Inlining Heuristic in the Glasgow Haskell Compiler

Drawing upon the information learned in Chapter 4, this chapter attempts to improve GHC's inliner through an experimental, empirical approach. This chapter presents experiments which attempt to redesign the inliner to achieve the maximum-observed speedup seen in Chapter 4, without the aid of the package developers' inlining pragmas.

The contributions of this chapter include the exploration of three experimental approaches to improve GHC's inliner using machine learning: a genetic algorithm, neural networks, and graph neural networks. Although the experiments do not produce significant performance speedups, they provide useful insights for the solution provided in Chapter 6.

Section 5.2 introduces a genetic algorithm. Because of GHC's compile times, we observe that we must either try another approach or attempt to seed the population with pre-trained neural networks, which we introduce in Section 5.3.

In Section 5.3, we attempt to train neural networks over the benchmark packages in Chapter 4, as we can produce labeled training data of ideal inlining decisions by setting GHC's magic numbers (as discussed in Chapter 4) to the optimal values preferred by each individual package. We see, however, that although we can train the neural networks to high accuracy, we cannot achieve the same compile times seen in Chapter 4. We observe that there is a tiny fraction of training data which has identical features but also has both 0 and 1 labels, suggesting that correctly making a very small fraction of decisions heavily determines performance outcome.

In Section 5.4, we approach the problem instead at the source-code level, to predictively place inlining pragmas at function declarations. We attempt to collect training data by adding or removing individual pragmas and noting run-time differences. Despite high training accuracy again, no significant speedups are observed—even when overfitting and predicting on the same package.

Exploration of why these attempts failed leads to the insights explained in Section 5.4.7. These insights will motivate the hot call chain approach described in Chapter 6, a technique that successfully achieves a significant speedup by inlining along call chains of hot functions identified by profiling.

5.1 Introduction

In the previous chapter, we explored the space of the Glasgow Haskell Compiler's inlining decisions by parameterizing and modifying its hand-coded constants, or magic numbers, at compile time.

Through an iterative search of these randomized numeric parameters, we found that each of 10 packages selected from Stackage was able to achieve a significant speedup—3% or faster execution time, compared to GHC's default inlining heuristic. However, we observed that no single configuration of magic numbers achieved the maximum-observed mean speedup of 27%, and each package preferred its own unique set of magic numbers. Further, the maximum observed mean speedup of any configuration was only 7% without the inclusion of developers' inlining pragmas in the packages' original source code.

The last chapter's results concluded that because the magic numbers could not be modified to one configuration that gives a speedup near that of the best-case speedup for all packages with the help of developer pragmas, the magic numbers themselves were likely not capturing the right information to make more optimal decisions. Therefore, a redesign of the inliner is warranted.

5.2 Training an Inliner from a Genetic Algorithm

5.2.1 Motivation

Work by Cavazos and O'Boyle [26] showed promising results for improving the inlning heuristic in the Jikes JVM compiler through the use of a genetic algorithm. Additionally, some research suggests that genetic algorithms could offer a better solution than neural networks because they are resistant to problems faced from gradient descent [155]. This section explains an attempt to use a similar approach to construct a machine-learning-based model for GHC's inliner using a genetic algorithm.

This attempt to modify the inlining heuristic in GHC aimed to replace the decision point in the Simplifier where the compiler decided whether or not to inline items with an `UnfIfGoodArgs` unfolding "`guidance`". An unfolding guidance of `UnfIfGoodArgs` is attached to normal identifiers [128], or items upon which the inliner will calculate its size-related cost/benefit analysis to decide whether to inline. For more information about GHC's inlining `guidance`, see Section 2.2.4.

Inlining Type	Total Inlinings	Percent
`UnfIfGoodArgs`	7124938	82%
`UnfWhen`	1281453	15%
`UnfNever`	301756	3%

Table 5.1: Type of non-trivial inlinings when compiling Cabal the Library.

To approximate the percent of non-trivial inlinings accounted for by the `UnfIfGoodArgs` guidance, we compiled Cabal the Library and recorded counts of all non-trivial inlinings which passed through the Simplifier for inlining consideration. As explained in 2.2.3, Cabal is the system that builds and packages Haskell libraries and programs, and *cabal-the-library* is the code which carries out that building and distribution. Table 5.1 shows that `UnfIfGoodArgs` makes up a substantial 82% of *cabal-the-library*'s non-trivial inlinings.

5.2.2 Formulating the Problem

Inlining in GHC may be formulated as a reinforcement learning problem, where an agent interacts with its environment in an iterative process to maximize a reward signal towards some objective [116]. GHC is the agent, the environment is the compilation, and the reward signal in this case is the measured run time of the compiled packages. Research suggests that for reinforcement learning problems, genetic algorithms may be a competitive alternative to the use of deep neural networks in cases where tasks have sparse or deceptive reward functions [155]; and additionally, work by Cavazos and O'Boyle [26] had already used a genetic algorithm for inlining for the Jikes RVM with promising results. We therefore proposed to construct a replacement inliner for non-trivial unfoldings (that is, inlining problems which could not be definitively decided before application of GHC's inlining heuristic) using a genetic algorithm.

5.2.3 Method

We aimed to replace a crucial part of GHC's inliner with a neural network, produced from a genetic algorithm, which would take a feature vector as input and then output a binary decision to inline or not. Specifically, we used a genetic algorithm called NEAT Python [150, 151], and feature vectors included features from the Magic Number work described in Chapter 4.

Because we would be replacing the cost-benefit calculation of the inlining heuristic in the middle of the compiler, our new model would be fired potentially millions of times per each package compilation. Making the models as small as possible would help keep compilation times under control, and it was therefore also desirable to use the smallest set of features possible—additionally to control for multicollinearity.

Package	**Total Inlinings**
reinterpret-cast-0.1.0	361,600
loop-0.3.0	363,940
metrics-0.4.1.1	355,448
set-cover-0.1	414,699
midi-0.2.2.2	713,076
nonempty-containers-0.3.3.0	920,901
ListLike-4.6.3	910,967
monoid-subclasses-1.0.1	954,442
hw-rankselect-0.13.3.1	1,166,781
poly-0.3.3.0	1,405,465
Average	**756,732**

Table 5.2: The total number of inlinings reported per package using GHC's default heuristics with no developer inlining pragmas.

Table 5.2 shows the total number of non-trivial inlining decisions recorded per package within the inliner within the Simplification pass in the middle of the compiler. The fewest recorded inlining decisions were 361,600 in `reinterpret-cast-0.1.0`, and the most were 1,405,465 in `poly-0.3.3.0`. Across the ten packages, there was an average of 756,732 inlining decisions per package.

The NEAT Algorithm

Overview. The NEAT algorithm produces populations of candidate genomes by breeding together a subset of genomes from the previous generation which are determined to be the best performers. Performance is determined by means of a fitness function that is defined for the given problem. The fitness function must return a single real number, and a hyperparameter can specify which function to use to judge it (e.g., maximum or minimum).

Genomes. A *genome* encodes the information for constructing a neural network, but it also contains information about the set of mutations that built these instructions and in what order the mutations were added. Genomes contain *node* genes, which contain information about neurons, and *connection* genes, which contain information about connections between neurons [121]. Table 5.3 shows the information contained in node genes, and Table 5.4 shows the information contained in connection genes.

Node Id	1	2	3	4	5
Node Type	input	input	hidden	hidden	output

Table 5.3: Information encoded in the node genes in NEAT.

Input Node	1	1	2	2	3	4
Output Node	3	4	3	4	5	5
Weight	0.2	-0.3	0.8	-0.1	0.5	0.2
Enabled	True	True	False	True	True	True

Table 5.4: Information encoded in the connection genes in NEAT.

The information in the node and connection genes can then be used to produce a neural network, which is also known as a "phenotype"—similarly to a phenotype in biology, which is a set of observable characteristics arising from an organism's genotype and environment. The neural network, or phenotype, composed from Tables 5.3 and 5.4 is depicted in Figure 5.1.

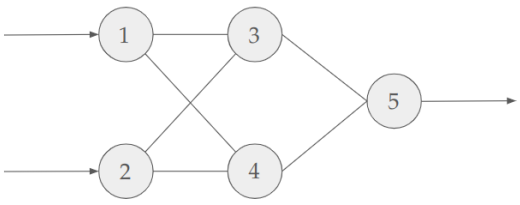


Figure 5.1: The network described by the nodes and connections encoded above in Tables 5.4 and 5.3, also known as a "phenotype" in NEAT.

Populations. The first population is constructed with a set of random features according to given hyperparameters. The NEAT algorithm starts with a minimally small architecture which will be built upon via the introduction of mutations in later generations. For the sake of simplification, we set the starting neural networks to all be feed-forward neural networks with 25 hidden nodes, none of which were initially connected, and allowed the algorithm to only manipulate connections between the nodes—e.g., enable connections, disable connections, and perturb connection weights. This is architecturally the same as building a neural network from scratch but limiting the hidden layer to a potential growth to 25 hidden nodes.

It is difficult to estimate a priori how large to make a neural network to maximize classification accuracy [94]; however, some rules-of-thumb are considered acceptable estimates. If the data is not suspected to be linearly separable, then there should be at least one hidden layer. If there is a relatively small number of features (as opposed to thousands or millions), then one or two hidden layers are acceptable. The number of hidden neurons may typically be less than the number of inputs and greater than the number of outputs. Also, the number of hidden neurons may be about 2/3 the number of inputs [73, 137].

Evolution. Genomes are changed across generations by *mutation* or *crossover*. In mutation, attributes of the genes in a genome may be randomly changed according to likelihoods assigned in the configuration as hyperparameters. This includes perturbation of connection weights, addition of nodes, and activation or deactivation of connections. For crossover, two genomes are combined together—where genes that have the same key are combined and genes that do not have common keys across the two genomes are copied from the higher-performing parent.

Termination. The algorithm runs until a member of one of its populations exceeds a given success threshold or for the specified maximum number of generations (if the success threshold is not met).

5.2.4 Design

The NEAT algorithm produces candidate inliner models at every generation, where the models are binarized and passed to a modified GHC as a custom flag. Inside the Simplifier at the inlining decision point, a function called `callSiteInline`, where GHC decides how to proceed with inlining depending upon unfolding guidance, passes features to the model and receives boolean decisions to inline or not. Genomes for the candidates whose inlining decisions result in faster test run times are then used as blueprints to build the next generation, by mixing pairs of genomes together and introducing random mutations.

5.2.5 The Inlining Decision Features

Table 5.5 recapitulates the features derived from Chapter 4, which represent a combination of attributes about each inlining decision's function body and context. We restarted the genetic algorithm multiple times for debugging purposes and to add additional features, shown in Table 5.6, in an attempt to provide the model with enough data to speed up learning. These additional features were readily available in GHC's Simplifier at the point of the calls to the machine learning model. Their addition did not produce a significant effect.

5.2.6 Training the Genetic Algorithm

The fitness function for each generation calculated the geometric mean speedup across all the packages, for each genome. The speedup referred to wall clock execution time of tests specified in the packages' cabal files. As in Chapter 4, random generation of tests was made consistent by setting the random seed in QuickCheck to be the same number across all runs.

At the beginning of each generation, the best performers from the previous generation were "bred" together to make the next generation, using a survival threshold of 20% of the previous population. Several such hyperparameters can be specified in the genetic algorithm, some of which appear in Table 5.7

As previously mentioned, 25 hidden nodes served as a maximum value of hidden nodes, as connections to them are disabled at the beginning of the run of the genetic algorithm; so in practice, the total number of hidden nodes would be some value between the number of inputs and the number of outputs. The population size of 100 is an acceptable value, as a rough estimate for recommended population size is commonly somewhere between 50 and 200 [1]. Although the fitness threshold was set to terminate at a 43% mean speedup, mean total speedup was reported at the end of each generation; thus, the run of the algorithm could be monitored for plateaued performance. Because addition of new hidden nodes was disabled, we made the probability of adding or deleting a connection relatively high, at 50% each, and additionally set the weight mutation rate high at 80%. The survival threshold, or minimum fitness to allow individuals to survive to the next generation, was left at the default of 0.2.

The genetic algorithm was trained over 20 packages, including 8 of the 10 packages from Chapter 4: `nonempty-containers-0.3.3.0`, `hw-rankselect-0.13.3.1`, `loop-0.3.0`, `set-cover-0.1`, `poly-0.3.3.0`, `metrics-0.4.1.1`, `ListLike-4.6.3`, and `reinterpret-cast-0.1.0`. These packages are listed in Table 5.8, and they were selected based upon having a test run time greater than 5 seconds and a successful compilation that took less than 5 minutes.

Abbr.	Value	Description
Continuation Features		
CSt	Int	Stop[e] = e
CCI	Int	CastIt co K)[e] = K[e ‘cast‘ co]
CAV	Int	(ApplyToVal arg K)[e] = K[e arg]
CAT	Int	(ApplyToTy ty K)[e] = K[e ty]
CSe	Int	(Select alts K)[e] = K[case e of alts]
CSB	Int	(StrictBind x xs b K)[e] = let x = e in K[\xs.b] or equivalently = K[(\x xs.b) e]
CSA	Int	StrictArg (f e1 ..en) K)[e] = K[f e1 .. en e]
Argument Features		
AINT	Int	The argument has structure.
AIVA	Int	The argument is a constructor application, partial application, or constructor-like.
AITA	Int	The argument is not interesting, i.e., deserves no unfolding discount.
ARGS	Int	The number of arguments
Expression Features		
LVAR	Int	Indicates if the expression is a lone variable.
IFV	Int	Number of vars that don’t occur in a coercion.
IFJI	Int	Number of join variables
IFL	Int	Number of literals
IFCase	Int	Number of case expressions
IFR	Int	Number of recursive lets
IFLNR	Int	Number of non-recursive lets
IFCast	Int	Number of cast expressions
IFLam	Int	Number of lambda abstractions
IFApp	Int	Number of applications
FOLDTY	{ 0, 1 }	Type of unfolding: UnfWhen or UnfIfGoodArgs
SIZE	Int	The size of the expression
WF	{ 0, 1 }	GHC estimate if expr. will not duplicate work

Table 5.5: Collected IR data: their abbreviations, possible values, and descriptions.

Abbr.	Value	Description
enough_args	{ 0, 1 }	The number of arguments is $\geq$ the unfolding arity
boring_ok	{ 0, 1 }	The result of inlining the expression is no bigger than the expression itself
uf_arity	Int	The unfolding arity
cont_info_ftr	{ 0, 1 } $\times 7$	A one-hot encoding indicating whether the context type was `BoringCtxt`, `CaseCtxt`, `ValAppCtxt`, `DiscArgCtxt`, `RuleArgCtxt`, `RhsCtxt`, or other

Table 5.6: Additional features for the genetic algorithm.

Hyperparameter	Value
Hidden nodes	25
Population size	100
Fitness threshold	1.428
Fitness criterion	max
Add connection probability	0.5
Delete connection probability	0.5
Connection weight mutation rate	0.8
Survival threshold	0.2

Table 5.7: Some of the hyperparameters used in the genetic algorithm that produced candidate models to make inlining decisions. Exhaustive hyperparameters are specified in a configuration file.

5.2.7 Performance of the Genetic Algorithm

Compile time in GHC is currently notoriously slow [89], and poor inlining heuristics makes them even slower. For the genetic algorithm, timeouts for compilation and testing were set at 30 minutes each, so the worst-case time necessary for each generation was about $30 + 30 + ((30 * 5)) * 20)$, which accounts for one compilation and one test run to be discarded (in case it contained any additional test compilation)—which could be built in parallel—and five test runs, which must be run sequentially and in isolation for each package on each genome. This worked out to about 3,060 minutes, 51 hours, or 2.13 days to run each generation with a population of 100. Larger populations are preferable to raise the chance of learning a good model, and the number of generations necessary to produce a result through a GA may vary widely, from 100 to hundreds of thousands of generations [132]. It could have realistically taken a year to obtain a result from the GA, assuming the algorithm did not have to be debugged and restarted.

Package name	Default time
ListLike-4.6.3	51.809
RSA-2.4.1	45.835
Ranged-sets-0.4.0	10.351
combinatorial-0.1.0.1	24.846
exp-pairs-0.2.0.0	97.630
hsini-0.5.1.2	25.924
http-conduit-2.3.7.3	5.299
hw-rankselect-0.13.3.1	13.196
json-rpc-1.0.1	98.818
loop-0.3.0	44.245
metrics-0.4.1.1	79.729
nonempty-containers-0.3.3.0	7.129
oauthenticated-0.2.1.0	11.558
poly-0.3.3.0	220.133
ramus-0.1.2	7.570
regex-applicative-0.3.3.1	26.639
reinterpret-cast-0.1.0	59.815
replace-megaparsec-1.2.0.0	8.063
scientific-0.3.6.2	41.428
set-cover-0.1	34.622
throttle-io-stream-0.2.0.1	15.470
x509-validation-1.6.11	14.662

Table 5.8: Packages used to train the genetic algorithm.

Table 5.9 shows increasingly higher percentages of successfully compiled packages at each new generation, which is one metric of improvement. Better inlining strategies are correlated with faster compile times, and poor inlining strategies can often cause compilation to diverge.

Generation	Mean Speedup	Successfully Compiled
1	0.16	66.7%
2	0.27	90.5%
3	0.28	83.2%

Table 5.9: The mean speedup and percent of packages successfully compiled within each generation of the genetic algorithm before it was stopped.

Additionally, Figure 5.2 shows that the algorithm seemed to be learning and making an improvement in terms of execution times, as evidenced by the recorded mean speedups of each genome. Across the three generations depicted, the mean of the geometric mean speedups across all genomes increased with each subsequent generation. However, the third generation was still 72% slower than default GHC—implying the need for several more generations just to have a mean compilation time comparable to default GHC. With the experiment requiring so much time to conduct, it was necessary to look for faster ways to learn a new inlining strategy. Section 5.3 discusses an approach which seeds the initial population with neural networks which have already been trained rather than starting with minimal random architectures.

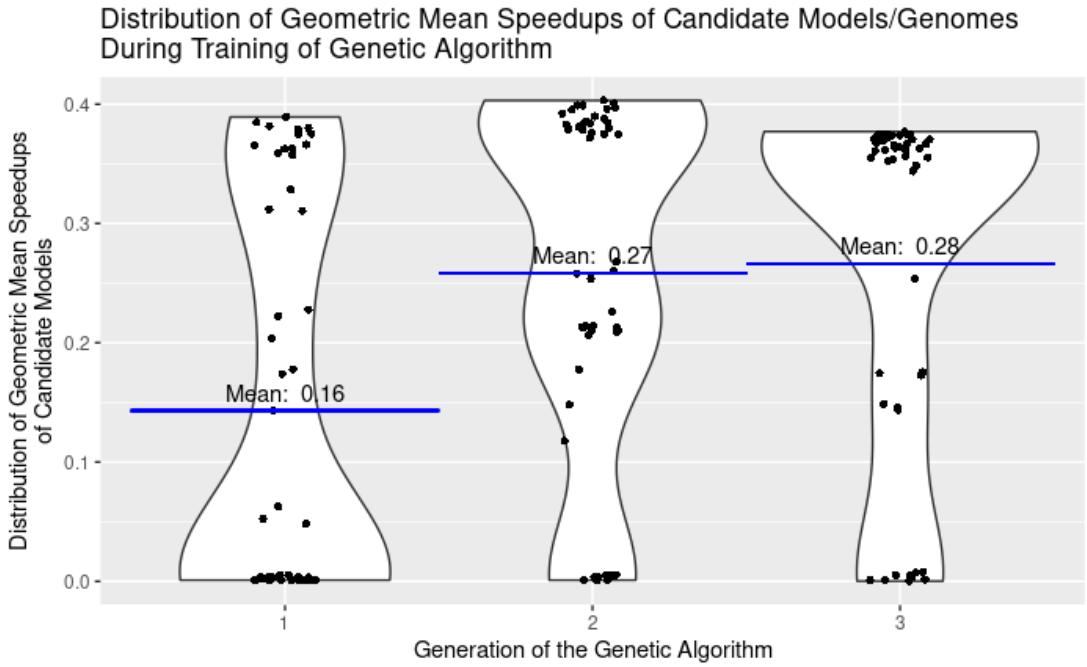


Figure 5.2: Genomes' execution times per generation as violin plots with jitter.

Although it would have been possible to continue running a genetic algorithm with more time and/or compute power to see whether a better inlining heuristic than the default could be achieved, ultimately its success seems unlikely because of the matters discovered and discussed in Section 5.5.

Speeding Up the Genetic Algorithm with a Seed Population

To address the problem of slow training time using the genetic algorithm, we decided to train an initial population of neural networks with gradient descent over the individual packages. This effort is described in the next section, Section 5.3, and the intent was to breed the per-package trained neural networks together using the existing genetic algorithm to create a generalized classifier which would work over the given and unseen packages.

5.3 Training ANNs to Predict Inlining from Best-Case Magic Numbers Training Data

5.3.1 Motivation

Following the attempt to improve inlining with a genetic algorithm which produced neural networks randomly from scratch, as described in Section 5.2, it seemed reasonable to instead try to use—or at least start with—a supervised learning approach. The next attempt aimed to create labeled training data which showed good instances to inline based upon the performance improvements demonstrated in Chapter 4. This training data could then be used to train a supervised learning model.

5.3.2 Overview

The plan hypothesized that by using the magic number parameters presented in Chapter 4 compiled over the set of packages with their pragmas included, as this combination produced the largest observed speedup, feature vectors at the point of each inlining decision could be emitted along with GHC's decision of whether or not to inline. These sets of feature vectors and decision labels could then be used to train one or more deep neural networks to output the correct set of decisions required to compile the aforementioned packages with the maximum-observed speedup; and thereafter, the network(s) could be tuned to generalize to other packages.

5.3.3 Production of the Labeled Training Data

GHC was modified to output the numerical values of each feature listed in Section 4.11 at every decision point to inline an item at a call site in the Simplification pass. Using Configuration 265 from the work performed in Chapter 4, for which `poly-0.3.3.0` had a 315% speedup with pragmas, and compiling `poly-0.3.3.0` with its developer pragmas, 2,071,017 inlining decisions were recorded along with their accompanying features. A summary of Configuraton 265 can be seen in Table 5.10.

Feature	Value		Feature	Value
FI	1		AC	17
AA	17		FFD	17
NTA	11		FDD	11
DAC	40		CB	40
RAC	60		CA	60
RC	19		BA	19

Table 5.10: Configuration 265 Features and Values

To recapitulate, Configuration 265 was the single best configuration across the 10 benchmarks from Chapter 4. Its speedup for set-cover was additionally 29%, whereas the best-observed configuration for set-cover was 30%; and for monoid-subclasses, this configuration produced a 6% speedup compared to the package's best-observed speedup of 7%. Therefore, this single configuration could be used to produce nearly optimally labeled data for three packages.

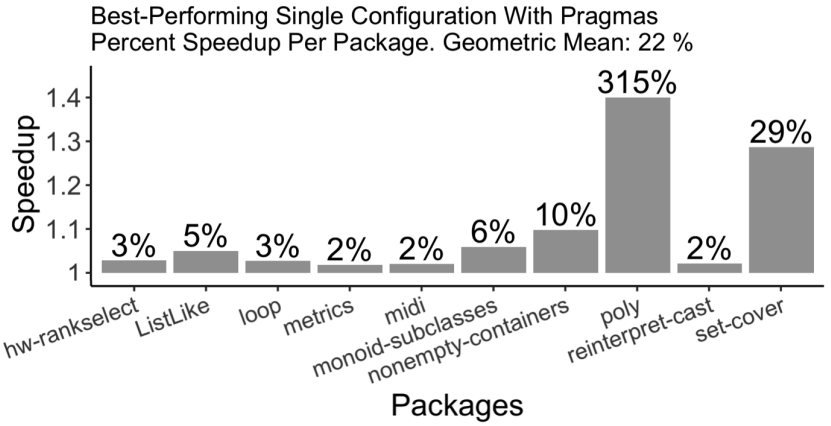


Figure 5.3: Best-performing single configuration of magic numbers in the GHC inliner across 10 benchmarks, with developer inlining pragmas.

Imbalance of Classification Labels

Examining `poly-0.3.3.0`, Configuration 265 produced 2,071,017 non-trivial labeled inlining decisions for that package. Of those labeled decisions, GHC decided not to inline 1,953,547 items and to inline 117,470 items. That means 5.7% of the considered items were inlined. To account for the sparsity of inlined data points versus non-inlined data points, we used oversampling to correct the imbalance of classes in the training data.

5.3.4 Model Construction

Each model was a feedforward neural network with 25 hidden nodes with ReLU activations and a sigmoid activation output. Training used binary cross entropy loss with the Adam optimizer set at a learning rate of .01. The models trained for 80 epochs or a plateau in learning.

The weights from the trained neural networks could be transferred to the initial population of neural networks in the genetic algorithm via a script.

5.3.5 Training Accuracy and Performance

The trained model predicted when not to inline, according to this compilation of `poly-0.3.3.0` from GHC, with 99.6% accuracy over the do-not-inline data. For when to inline, the model predicted with 93.6% accuracy over the do-inline data. These numbers are depicted in Table 5.11.

GHC Not Inlined	1,953,547		GHC Inlined	117,470
Predicted Not Inlined	1,945,861		Predicted Inlined	109,905
Accuracy	99.6%		Accuracy	93.6%

Table 5.11: Inlining Prediction Accuracy

Despite this, when plugging the model's weights into GHC, compilation and run time were orders of magnitude slower than GHC's default decisions.

As another example, we attempted to explore the neural architecture on `set-cover-0.1` alone because its compile and run times were relatively small and, as aforementioned, the same magic number configuration produced a near-optimal speedup for it. The `set-cover-0.1` package had a default run time of 15.485 seconds. The altered configuration of magic numbers for `set-cover-0.1` without developer pragmas gave a run time of 13.185 seconds; however, the training data generated from compilation of that configuration gave a run time of 17.715 seconds—a 12.6% slow down. This result came from a neural network with 35 hidden nodes and trained for 150 epochs.

As a sanity check, we plugged a "coin-toss" inliner into GHC to randomly choose whether or not to inline with a 50% probability. For the random inliner, compilation time on `poly-0.3.3.0` timed out after 10 minutes, preventing a collection of test time. On `set-cover-0.1`, compilation took 1 minute 14 seconds, compared to 37 seconds default; then, it timed out after 10 minutes for testing.

Ambiguously Labeled Training Data

Upon investigation, and as seen in Table 5.12, 2.88% of the data had conflicting labels. That is, 2.5% of the data which had a No label also had a Yes label, and 9.4% of the data which had a Yes label also had a No label.

GHC No-Inline Conflicting Labels	48,526		**GHC Inline Conflicting Labels**	1,1070
Percent From “No”	2.5%		Percent From “Yes”	9.4%
Percent From Total	2.3%		Percent From Total	0.53%
Total conflicting data in dataset: 59,596 labels				2.88%

Table 5.12: Breakdown of Conflicting Inlining Labeling. Conflicting datapoints in the training data have identical values for each of the collected features yet both Yes and No for the classification to inline.

In cases where the training data had multiple possible labels for the same feature vectors, it may be concluded that the chosen representation of features is not descriptive enough. It may be possible to adjust the features to try to increase accuracy; however, given that the total percent of mispredicted data which should be Yes and was instead No was merely 0.53% of the dataset (not even 1%), it seemed apparent that approaching the problem differently would provide a solution faster than trying to further increase accuracy.

It is also possible that some amount of the data with conflicting labels came from duplicated or sufficiently similar code where one or more duplicates have an inlining pragma and one or more duplicates do not. In this case, we presumably want to inline the duplicates as well. Section 5.4.8 provides an observation as to what may happen in that event. In summary, it leads to over-inlining in the packages examined.

5.4 Using Graph Neural Networks to Predict Pragma Placement in Haskell Source Code

5.4.1 Motivation

Rather than attempting to learn from every decision made inside the inliner, the next approach tried to determine when it would be advantageous to place an inlining pragma directly into source code, at the place of the function declaration. Evidence from Chapter 4 suggested that well-informed inlining pragmas *may* induce significant speedups in some situations, so perhaps it could be possible to learn when this is the case? Additionally, there would be no

significant additional compilation time inside GHC with this approach, because it would not change the decision process for every non-trivial inlining candidate. Further, programmers could potentially review pragma recommendations made by the model before proceeding with compilation if they were placed directly back into the source code.

A program may be conceptualized as a graph, and it will be parsed as such at the beginning of compilation. Using a graph neural network over a graph representation of the program in combination with a convolutional neural network (CNN) would enable the CNN to recognize multi-scale, hierarchical features within the code [188]. “Multi-scale” means a feature can be recognized regardless of its topological scale, and “hierarchical feature” learning is the learning of features of varying levels of complexity in different layers of the CNN. Further, a graph neural network could reason about the program as a whole and make decisions about inlining once at any function declaration, rather than multiple times through repeated passes in the Simplifier.

5.4.2 Overview

Chapter 4 showed that although it was possible to see dramatic speedups from the use of inlining pragmas, examples of packages with such pragmas were rare despite the extensive use of inlining pragmas in the Haskell community’s central package archive, Hackage. The first experiment in this section aims to identify pragmas that have any measurable benefit at all and train a model to place them in similar places across multiple packages, as described in Section 5.4.4. An even simpler attempt in Section 5.4.7 tries to look at the notable success of pragmas in `poly-0.3.3.0`, the package from Chapter 4 with by far the greatest observed speedup from developer pragmas, and simply predict where the developers of that package may have placed pragmas in other packages at similar points in code to see if any effect could be observed. Finally, Section 5.5 gives some insights into why these approaches fail to produce a speedup.

5.4.3 Setup: Graph And Model Construction

Graph neural networks make predictions over data which is encoded as a graph. In the case of a program, this could reasonably be a control-flow graph. At the time of these experiments, the version of GHC used had no instrumentation to output an interprocedural control-flow graph.

Although we tried to find an existing tool to produce flow graphs, few options were available. The leading candidate, graph-trace, produced a 36 GB graph for `set-cover-0.1`. That package had about 2,781 source lines of code (SLOC), which was about the median SLOC among the 10 packages in Chapter 4. Graphs of this size would have been too large to work with; so as a proxy, we decided to use abstract syntax trees (ASTs) produced during compilation for each of the modules, which can be encoded as graphs.

Constructing the ASTs: A Simple Example

Figure 5.4.3 presents a very small 7-line Haskell program. Following the renamer pass, GHC produces an AST which is over 279-lines for this simple program, as presented in Appendix B. (The AST in Appendix B is slightly modified to be more easily parsed in the experiments, so the unmodified AST dump from GHC would be longer.)

```
1       addexclaim :: String -> String
2       addexclaim name = name ++ "!"
3
4       main = do
5           foo  <- putStrLn "Hello, what's your name?"
6           name <- getLine
7           putStrLn ("Hi, " ++ addexclaim(name))
```

Figure 5.4: A small 7-line program written in Haskell.

A simplified graph of the AST for the body of the `addexclaim` function is displayed in Figure 5.5. For any realistic package, processing the program as one single graph would be intractable because of the size of the ASTs.

Graphs are instead created for individual functions. Line 11 of Appendix B indicates a `FunBind`, or a function binding for a function body with its name, "Main.addexclaim" on line 15 of the AST. The `FunBind` indicates the root, or beginning, of the graph of the function body. Additionally, the function "addexclaim" has a type signature, which is emitted starting on line 223 of the AST with `TypeSig` followed later by the function's name on line 226.

AST Graph Composition

For each function, we created a graph of the function's abstract syntax tree; and if present, we created a graph of the function's type signature as well. Following collection of all function and type signature graphs, we combined each function's body and type signature together into one graph for each function.

We parsed the ASTs of every source file in the packages to collect the names of all present syntax features. There were 300 total, as shown in Appendix 2. Some examples of these features include `ClassDecl` for a class declaration, `BangPat` for a bang pattern, and `StringLiteral` for a string literal. For variable values like fast strings and numbers, we changed the parser to emit a constant value. For example, where a fast string may otherwise have been `FastString: "Hello World"`, we made the parser output `FastString: "FASTSTRING"`. See line 57 of Appendix 1 for an example.

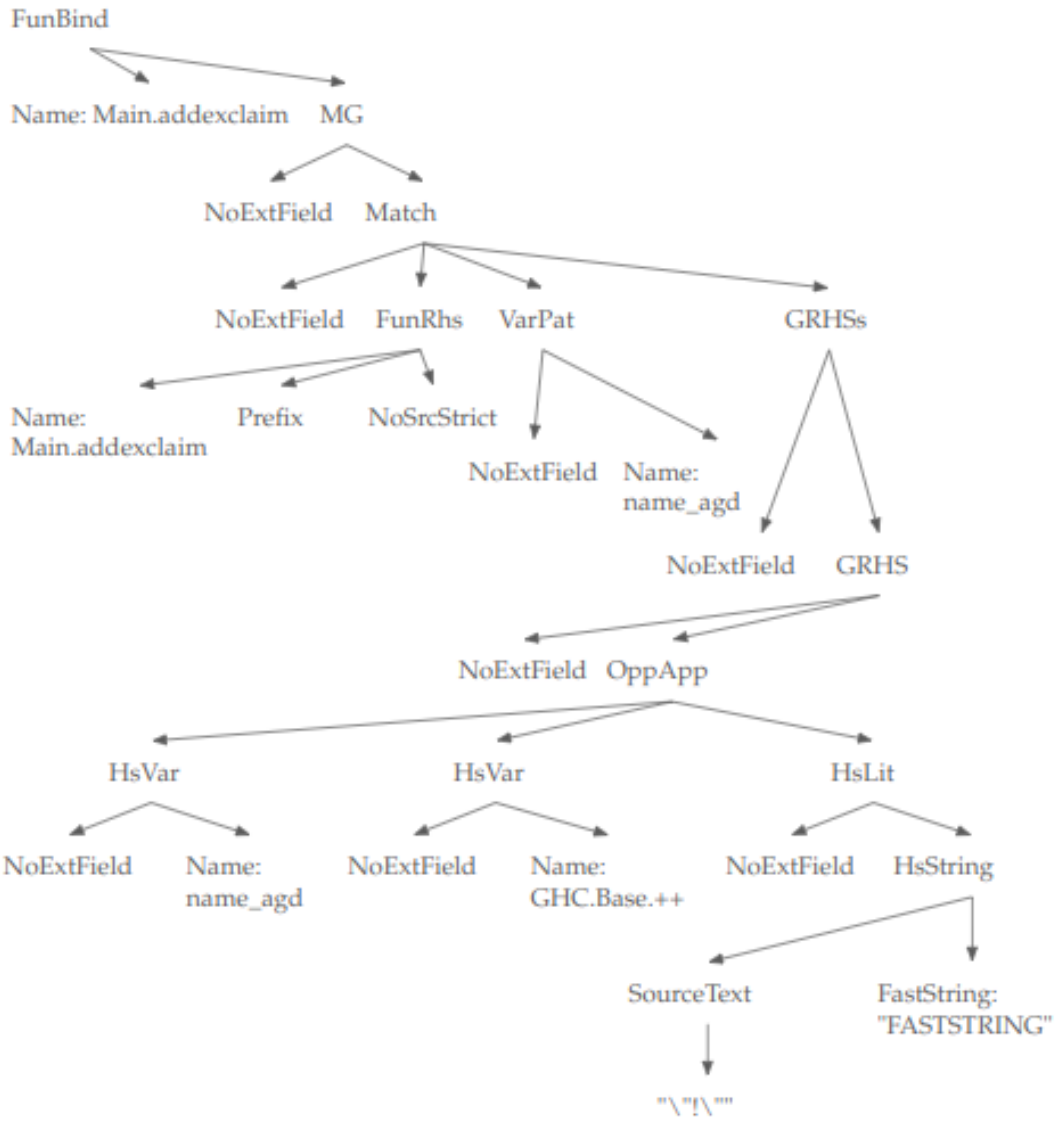


Figure 5.5: A simplified representation of the function `addexclaim` in 5.4.3.

5.4.4 Training a Model Over Pragmas with Verified Performance Benefit

Production of Labeled Training Data

The packages used to train the graph neural network were a different selection from those used for the genetic algorithm or the ANNs trained over the altered-magic-number data in the previous sections. For this experiment, we prioritized packages that had the largest number of pragmas possible to maximize the likelihood of having enough training data; however, the

packages also had to build and test successfully on the machine and have tests that ran for some reasonable amount of time. Although we tried to select packages with tests that ran for over 1 second, some exceptions were made for packages that had a high number of inlining pragmas.

Here, training data would be composed of graphs representing functions with inlining pragmas attached to them which have a verifiable significant effect on performance. Table 5.13 shows the package names, the count of the inlining pragmas inserted by the packages' developers, and the minimum time recorded from running the package's tests with their pragmas removed.

Package name	Number Pragmas	Default Time with No Pragmas
mono-traversable-1.0.15.1	750	205.706
nonempty-containers-0.3.3.0	520	4.550
folds-0.7.5	341	4.788
storablevector-0.2.13	298	4.280
intervals-0.9.1	213	24.606
paripari-0.6.0.1	190	5.707
persist-0.1.1.4	175	0.940
vector-algorithms-0.8.0.3	146	23.425
bitvec-1.0.2.0	66	3.225
poly-0.3.3.0	57	205.706
formatting-6.3.7	44	0.365
string-transform-1.1.1	36	1.051
set-cover-0.1	16	34.027
xeno-0.3.5.2	12	0.363
carray-0.1.6.8	11	3.250
classy-prelude-1.5.0	11	1.160
metrics-0.4.1.1	9	88.075
xlsx-0.8.0	5	7.020
SHA-1.6.4.4	5	1.457
data-interval-2.0.1	2	1.737

Table 5.13: Packages used to produce graphs to train the graph neural network.

When significance was measured by adding pragmas one at a time, this set of 20 packages produced 1118 significant data points, where "significance" could be either a slow-down or speed-up. Of these significant datapoints, 669 produced a speed-up and 449 produced a slow-down.

5.4.5 Model Training

Following some manual hyperparameter tuning, we used a graph convolutional network model with 64 embedded features, 450 hidden nodes, batch size of 250, 2 convolutional layers, global mean pooling, a linear output layer, and the Adam optimizer with a learning rate of 0.0001. Models were trained to a plateau of prediction accuracy with a set limit of 400 epochs.

5.4.6 A Naive Approach: Train by the Measured Benefit of Individual Inlining Decisions

Two methods were used to produce training data to predict when to inline: measurement of performance difference upon removal of a pragma and measurement of performance difference upon addition of a pragma. The second was attempted upon failure of the first.

Performance Difference With Pragma Removal

The first line of reasoning postulated that removal of an inlining pragma at a point in code where inlining would be beneficial, but the compiler does not currently recognize it, would produce a measurable slow-down of execution time. Therefore, each `INLINE` or `INLINABLE` pragma was removed, one by one, with its resultant performance difference recorded.

For each pragma in the training data set, we removed the individual pragma, compiled, and collected five run-time samplings of the package's test execution. For default measurements, we left all pragmas in and collected five timings. We then checked whether any of the pragma removal timings overlapped with the default timings, labeling the pragma significant if they did not and the modified timings were faster than default. Figure 5.6 shows boxplots for timings collected on one pragma in the project `intervals-0.9.1`.

The test data was a set of 10 packages which were not used in the training set. The packages in the test set were: `alex-3.2.5`, `bv-little-1.1.1`, `hw-fingertree-strict-0.1.1.3`, `approximate-0.3.2`, `hyperloglog-0.4.3`, `happy-1.19.12`, `distributive-0.6.1`, `hw-fingertree-0.1.1.1`, `comonad-5.0.6`, and `bits-0.5.2`.

Of the 2,907 developer pragmas in the 20 training packages, 1,625 were determined significant with no times overlapping with default. These 1,625 timings trained models from 81% to 90% accuracy.

The models predicted pragma recommendations over the graph representations of functions extracted from the test set in CSV format, and the predicted pragmas were appended back into the source files.

Although there appeared to be no significant drop in performance from any possible over-inlining by adding pragma recommendations from the models, no significant positive change in performance was observed over any packages.

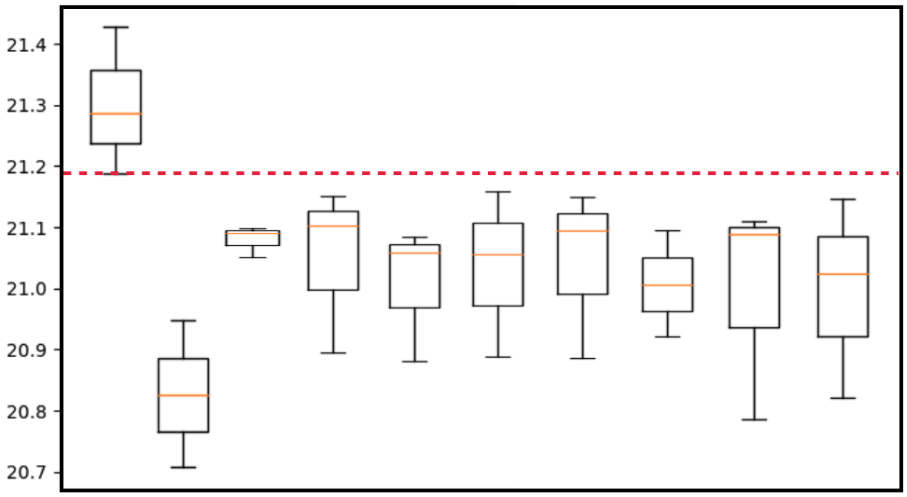


Figure 5.6: Boxplots of time collections for 8 pragmas determined to have significant speedups in the file Kaucher.hs for the project `intervals-0.9.1`. The red line indicates the value of the fastest default timing, or fastest time with no developer pragmas removed.

We created another set of training data which labeled datapoints significant one of the collected timings overlapped with the default timings—rather than no overlap. However, no significant differences in performance measurements arose from this data either.

Performance Difference with Pragma Addition

When the approach of training by pragma removal failed, a second round of data was collected by instead measuring the performance difference when all pragmas were removed and then each pragma was placed into the code on its own.

Attempts to learn an inlining policy with this data once again produced a high-accuracy model: the training reached 91% accuracy over the training datapoints; but when the model's pragma recommendations were deployed into compilation, package test time again achieved no significant measured speedup over default GHC.

Section 5.4.7 gives a more in-depth explanation of why these data sets failed to produce a good machine learning model for inlining.

5.4.7 Why Is Inlining So Hard To Predict? A Case Study

Motivation

One package in the Magic Numbers study, `poly-0.3.3.0`, achieved a 154% speedup just with the addition of well-placed inlining pragmas. Efforts to predict where to place inlining pragmas in `poly-0.3.3.0` failed with machine learning, even when the model was deliberately overfit. Training a graph neural network over `poly-0.3.3.0` achieved a 94.1% training accuracy; however, timing of its tests with the learnt model were no faster than running the tests with all inlining pragmas removed. An inspection of the pragma recommendations showed that a number of them had been correctly inferred, but not all. This makes sense when considering that code may be extremely similar or identical across modules, yet developers may choose to annotate only specific points of the code with inlining pragmas because they have some knowledge of how it will be used.

This section reveals findings from a close inspection of the pragmas in `poly-0.3.3.0` to understand why no attempts to produce a performance improvement from pragma prediction had thus far worked, despite good accuracy of models to emulate decision making of other successful methods.

When Does Inlining Really Matter?

To inspect `poly-0.3.3.0`, we tried various combinations of pragma removal to uncover which minimal set of pragmas were necessary to achieve the observed speedup. Despite the package containing 57 total pragmas, only 10 of them accounted for almost all of the performance gain. These 10 pragmas are presented in Table 5.14, and all 10 of them appear in one file.

Furthermore, the functions to which all of the effective pragmas were attached were linked to each other. Figure 5.7 roughly shows their call order, although arrows do not imply a direct call—other functions may be present in the arrows between.

The important point is that if any function hop between the roots—`convolution` or `karatsuba`—is either not inlined automatically by GHC or not explicitly inlined by the developer, then the speedup is not gained. Therefore, it is reasonable to hypothesize that in order to see the performance gain, these `convolution` and `karatsuba` functions need to be inlined all the way up to where they are originally called in the source code.

Function Name	Pragma Type
(*)	INLINE
times	INLINE
karatsuba	INLINABLE
convolution	INLINABLE
subst	INLINE
subst'	INLINE
substitute	INLINE
substitute'	INLINE
deriv	INLINE
deriv'	INLINE

Table 5.14: Pragmas in poly-0.3.3.0 which accounted for virtually all of the performance improvement. All of the pragmas occurred in the same module.

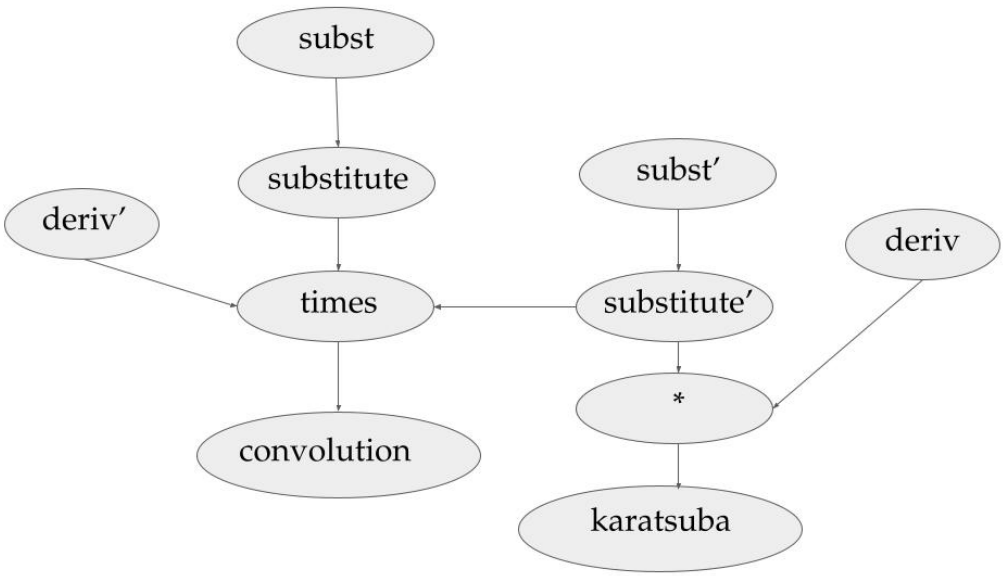


Figure 5.7: A call graph of the functions in poly-0.3.3.0 which must be inlined together to produce a speedup.

Analysis of the Data Collection: Why Performance Measurements Did Not Work

Using measured performance differences introduced by pragmas only makes sense when their beneficial effects are independent of other pragmas. In cases where this occurs, it is plausible that a comparison of a point in code where a pragma is added versus the same point in code where a pragma is removed would have significant and opposite performance

effects. When we examine the reported significance, by measurement, of pragmas in the file `Dense.hs`—which has the 10 significant pragmas mentioned in Section 5.4.7—none of the pragmas have both a positive observed effect when added and a negative effect when removed.

Table 5.15 shows the breakdown of types of contradicting data between the two methods of collection in the file `Dense.hs` where virtually all of the performance benefit from pragmas occurs. The cell “Significant and Opposite Effects Observed" shows that no pragmas had both a positive effect when added and a negative effect when removed. “Disagreement on Significance” shows that 22 pragmas were recorded to have a significant effect either positive when added or negative when removed, but not both. “Significant but Effects Contradictory” shows that no pragmas had a significant effect when added but a significant yet contradictory effect when removed (for example, a speedup when added but also a speedup when removed). Finally, “Removal and Addition Both Insignificant” showed that 10 pragmas in the file seemed to have no effect at all, after removal or addition.

Significant and Opposite Effects Observed	**Disagreement on Significance**
0	22
Significant but Effects Contradictory	**Removal and Addition Both Insignificant**
0	10

Table 5.15: A confusion matrix of reasons why measures of significance on the addition and removal methods of collecting training data contradict each other.

Table 5.16 shows that some of the significant pragmas in Table 5.14 showed a significant effect when added alone but no significant effect when removed on their own, as seen in the cell “Addition Positive but Removal Ineffective”. The 15 pragmas referred to in that cell are listed in Table 5.17; and it can be seen that the significant pragmas `times`, `karatsuba`, `subst'`, `substitute'`, `deriv'`, and `deriv` appear to have a small effect when added on their own.

As a sanity check, we manually inlined all of the pragmas which had a positive measurable speedup on their own. When compiled, the resultant binary had no significant speedup over default execution with no pragmas. Even if the model had been trained to 100% accuracy over the labeled data, it likely would not have learned anything of value.

Removal Positive but Addition Ineffective	Addition Positive but Removal Ineffective
3	15
Removal Negative but Addition Ineffective	**Addition Negative but Removal Ineffective**
0	1

Table 5.16: Inlining pragmas which significantly effected performance with their individual removal or addition—but not both.

Function Name	Pragma Type
(+)	INLINE
(-)	INLINE
one	INLINE
times	INLINE
dropWhileEnd	INLINE
karatsuba	INLINABLE
scaleInternal	INLINABLE
unscale	INLINABLE
subst'	INLINE
substitute'	INLINE
deriv'	INLINE
deriv	INLINE
integral'	INLINABLE
integral	INLINABLE
var	INLINE

Table 5.17: Functions with pragmas whose addition was measured to have a significant positive performance effect but whose removal had no significant effect.

Potential for the Use of Profiling

When we removed all inlining pragmas and profiled `poly-0.3.3.0`, the functions `subst` and `subst'` were reported as the two functions taking the most execution time, by far. Respectively, they ran for about 49% and 30% of the total time, each. Naturally, then, profiling seemed to be a reasonable method to identify bottleneck functions, which lends motivation for the profile-based approach presented in Chapter 6.

5.4.8 An Observation: Trying to Predict Where Developers Would Place Pragmas

We additionally wanted to predict where developers of one package would place pragmas in another package. In order to do that, however, we first had to train a model that could correctly predict where the developer pragmas had been placed in its own code. With pragma predictions from a graph neural network, these predictions could easily be inspected manually by placing them directly back into the source file to potentially provide interesting information.

Unfortunately, the work in Chapter 4 showed that developers only produced significant speedups on two packages, `poly-0.3.3.0` and `monoid-subclasses-1.0.1`. However, `monoid-subclasses-1.0.1` only produced a 3% speedup with 334 pragmas, which would make post hoc analysis difficult. Despite only having a 2% developer speedup, we included `set-cover-0.1` because it had a small yet substantial number of pragmas, shorter compile time, and substantial yet manageable number of source lines of code.

Manual Inspection of Pragma Predictions

One observation we made regarded the antipattern of copied, pasted, and modified functions appearing similar to the labeled significant functions—and additionally, code that simply looked very similar. Many of the functions labeled with an inlining pragma in the two packages appeared very similar to functions that had no pragmas. Even when training a package on itself and trying to predict its own pragmas, the model would sometimes fail to predict a pragma where it should have.

If we assume that inlining performance can be significantly improved by inlining entire call chains, then one option to overcome the duplicate/near-duplicate code issue was making the model more eager to inline (particularly by increasing the classification threshold) to ensure all of the necessary functions received a pragma. Lowering the model's classification threshold could accomplish this by increasing the likelihood of the significant target function receiving a pragma, but would incidentally inline several additional similar functions and trigger a slowdown. Potentially, this issue may possibly also have been addressed by trying to make the features richer, but many functions present in the chains did not appear very syntactically unique in comparison to others. For these packages, assuming high enough discrimination between decisions could not be reached, over-inlining was likely to occur.

Addressing Ambiguous Training Data and Developer Inlining Pragmas

Take, for example, the ambiguous training data in Section 5.3.5. If we assume it is caused by the presence of a pragma on a function with duplicates having no pragmas and choose to err on the side of the pragma, then we would decide to add pragmas to its duplicates. Our attempts to approximate that approach by decreasing the classification threshold produced either no speedup or an eventual slowdown, as the classification threshold was decreased.

5.5 Conclusions

This chapter experimentally explored three machine learning approaches to improve GHC's inliner: a genetic algorithm, neural networks in the Simplifier, and graph neural networks to predict inlining pragma placement at the level of source code.

As a note, the most closely related work to the experiments in this chapter occurred in Trofin et al. [170], where the authors presented a framework to integrate machine learning—based upon reinforcement learning and evolution strategies—into the inliner for LLVM. The biggest difference between that work and the work done in this thesis is that LLVM is a low-level language, similar to assembly, which is not principally functional. As such, control flow information was likely more accessible. Additionally, the authors' optimization goal was binary size reduction rather than execution time reduction, where execution time reduction is harder or more time consuming to assess.

These experiments demonstrated that high prediction accuracy over labeled data does not translate to improved performance during compilation, implying that a tiny number of inlining decisions may have a drastic effect on performance. Additionally, reliable training data cannot be generated under the assumption that inlining decisions are independent of each other—such as by measuring the effect of one inlining decision change. The case study in Section 5.4.7 shows that actually, the relationship of functions with each other, and whether functions related to performance bottlenecks are *all* inlined, can decide performance improvement. For the package in this example, large performance improvements are observed when all functions along the performance bottlenecks are given an `INLINE` or `INLINABLE` pragma.

As explained in Section 5.4.7, effective inlining can be demonstrated when every call leading to a cost center is inlined. To find these call chains, we need some sort of flow analysis. When functions or procedures are not first-class, intra- and inter-procedural analysis is relatively straightforward. In functional languages, functions are first-class and can be passed as variables. This means that functions do not have clear predecessors or successors when examined statically. Furthermore, for first-class languages that have dynamic dispatch, which function is invoked will depend upon runtime values. Thus, determining control flow in functional languages is a difficult problem. This will be the motivation for the technique introduced in Chapter 6.

Chapter 6

Hot Call-Chain Inlining for the Glasgow Haskell Compiler

6.1 Overview

Chapter 5 presented evidence that performance could be improved by inlining every function inside connected call chains. An additional observation suggested that the call chains which incur the most run-time costs may be found by profiling. Profiling reports, however, will not fully tell us which functions call which other functions. How, then, do we uncover the call graphs of hot profiled functions and go about inlining the functions within them? As mentioned in the previous chapter, control flow is a hard problem in functional languages.

This chapter presents a profile-directed technique to direct inlining decisions in the Glasgow Haskell Compiler. We show that simply inlining "hot" functions, as revealed by profiling summaries, does not lead to significant improvement. However, inlining along the hot dynamic call graph is frequently beneficial. Due to the higher-order nature of Haskell, determining this call graph is non-trivial. We develop a technique to extract call chains of hot functions and leverage the Glasgow Haskell Compiler's existing functionality to safely influence inlining decisions through pragma placement along these chains.

We then show that hot call chain inlining yields a geometric mean speedup in run time of 9% over GHC's default inlining heuristics across 17 real-world Haskell packages. This method can be used in the presence of pre-existing developer pragmas to produce a mean speedup of 10% across those same packages. Furthermore, the hot call chain technique produces no more than a 1% mean code size increase across all packages, and no more than a 7% code size increase for any individual package.

Section 6.2 motivates the approach and provides an overview, giving an example of a call chain from real-world code and explaining the challenges of control flow in functional languages. Section 6.3 describes the methods used to filter profiling information, collect call graphs, and place inlining pragmas. Then Section 6.4 presents the tooling to binarize and

insert pragma recommendations and to collect profile information. Experimental setup is described in Section 6.5, including package selection criteria, handling of rewrite rules and randomly generated tests, response measurement, and description of each set of experiments.

In Section 6.6, results are presented for each set of experiments, along with results for an adjustment of significance thresholds for the profiling information, effects of hot call-chains on binary sizes, comparison against the magic-number approach presented in Chapter 4, and results for a change of input data (benchmarks versus tests). Of note, six of the packages used in Chapter 4 are included in the benchmark set for this chapter's experiments, but four of the packages are excluded because they do not meet selection criteria; however, they are included for completeness of comparison in Section 6.6.7. Finally, Section 6.7 concludes the chapter.

6.2 Introduction

Chapter 4 established that the lack of progress on GHC's inlining capabilities has led to Haskell programmers taking matters into their own hands. A simple string search showed that about 19% of all packages in Haskell's central package archive, Hackage [36], contain inlining pragmas. These pragmas may indicate programmers' attempts to make their code run faster than GHC's default inlining strategy by coercing GHC to inline specific, perhaps larger things than it otherwise would have. In practice, these pragmas often do not have much of a performance effect; but in a handful of cases, they demonstrate that well-placed pragmas can produce a significant speedup [77].

One approach to improve performance is to determine those sections of code that dominate execution time. If these sections contain function calls, then perhaps simply inlining the indicated functions from, for example, a profile summary may improve execution time without excessive inlining. In Section 6.6.1, however, we show that such an approach provides limited benefit.

We show that if we can further identify the call chain connected to the hotspot and inline all functions within that chain, it is possible to achieve significant performance improvement. However, determining such a call graph within Haskell is highly challenging, given its higher-order nature. The problem of determining program control flow to precisely retrieve all of these call chains, incidentally, is an exponential problem [108].

We demonstrate how inlining along full hot call graphs improves performance, and we present a conceptually easy approximate technique to do so. Additionally, we show that overapproximation is allowable in the specific case of inlining optimization. Although profiling hot functions is commonly used for optimization in other languages, it is not used for inlining in Haskell due to the difficulty of computing control flow for a call-by-need functional language.

We present a system which identifies profiling hotspots and cheaply approximates control flow to recover their related call chains, which we call *hot call chains* (HCC). We encode the names of the functions in these chains and pass them to a modified GHC that inserts the corresponding inlining pragmas along the recommended call chains. We then apply this technique to a set of 17 real-world packages, after removing developers' inlining pragmas, to we achieve a 9% mean speedup in run time over GHC's default inlining optimizations. Combining the hot call chain technique with existing developer pragmas brings the mean speedup to 10%.

6.2.1 An Example of a Case to Inline Call Chains

As an illustration of how inlining chains of functions related to hotspots can produce bigger performance improvements than simply inlining the functions identified as hotspots: In the package `set-cover`, we observed three hotspots in its test-suites and benchmarks for top-level functions declared in the file `Exact.hs`. Table 6.1 displays their names and percent time consumption, per a profiling report.

Function	Time (%)
intSetFromSetAssigns	7.6
updateState	6.9
step	4.2

Table 6.1: Hotspots listed in a profiling report in the file `Exact.hs` in `set-cover` and their associated time consumption.

Our tool uncovered the call chain associated with the hotspot `updateState`, which had a 6.9% reported time consumption in the profile summary. We present the functions related to `updateState` in a small graph in Figure 6.1 which has been extremely simplified. In this figure, function names with an asterisk (*) indicate that the package developers had already attached an `INLINE` pragma to the function declaration in the package's source code.

If we were to attempt the "naive" approach (described in Section 6.6.1) of simply attaching inlining pragmas to only the two hotspots in this graph, `updateState` and `step`, we would get about a 1% speedup. However, we observed the developers themselves added `INLINE` pragmas to much of the call chain on `partitions`, `search`, `step`, and `updateState`. These pragmas produce about a 6.6% speedup. If we also add pragmas to the other three functions in Figure 6.1, we then get a speedup of about 7.5%.

Our method uncovers another small chain in the same file:

$$\texttt{intSetFromSetAssigns} \rightarrow \texttt{mapIntFromSet}$$

Adding pragmas to these additional two functions then brings the total speedup for this file to 8%.

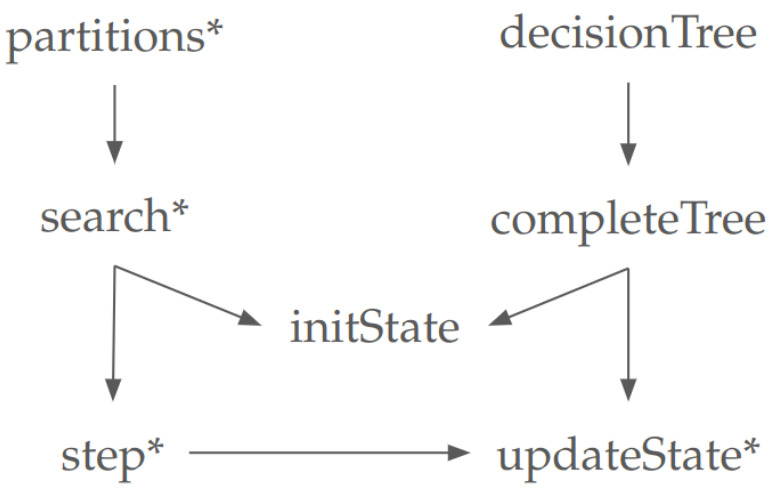


Figure 6.1: A simplified relationship of a group of functions in a file in `set-cover`. The functions `partitions`, `search`, `step`, and `updateState` are marked with an asterisk to indicate that the developers attached an `INLINE` pragma to them.

To describe it intuitively, a reported hotspot may indicate an entry point into a chain of functions, which can be conceptualized as nodes, connected in a path which calls down to one or more computationally intensive functions at the end of the path. In this case, it is often beneficial to inline every function in the path.

6.2.2 Challenges

Finding the call chains in bottlenecks requires some sort of flow analysis. When functions or procedures are not first-class, intra- and inter-procedural analysis is relatively straightforward. In functional languages, functions are first-class and can be passed as variables. This means that functions do not have clear predecessors or successors when examined statically [108]. Furthermore, for first-class languages that have dynamic dispatch, which function gets invoked will depend upon runtime values.

The first popular approach to the flow problem in higher-order languages was k-CFA for some value of k. 0-CFA is a context-insensitive (or mono-variant) analysis which does not distinguish different instances of program variables or points. Its computational cost is polynomial—cubic at best—and its computation is impractical for large programs [63]. Further, any k-CFA where $k \geq 1$ and a context-sensitive (poly-variant) analysis is performed is EXPTIME-complete [171].

6.3 Method

6.3.1 Profile Information

To collect profile reports for our 17 packages, we profile each target package's test suites and benchmarks, where these are executable targets marked `Test-Suite` or `Benchmark` in the package's `cabal` file. We use GHC's default threshold to identify hotspots taking either 1% or more of execution time or allocations. Those functions identified as hotspots indicate nodes in hot call chains, C. Given C, we determine all functions reachable on the call graph from C using the approximation described in Section 6.3.2 and attach inlining pragmas to them according to the system shown in Figure 6.2. In our implementation, specifically, the function names given are passed in as a binary, and GHC attaches the pragmas during de-sugaring of bindings.

6.3.2 Call Graph

As mentioned, determining actual control flow in Haskell is difficult due to its higher-order nature. To tackle this, we use static code structure as an over-approximation of dynamic control flow. We assume that if there exists a path in the abstract syntax tree (AST) from a function definition `a` to a call of another function `b`, there is a control flow between `a` and `b`. Determining this over-approximation is straightforward: we first examine the AST of each function `a` and record all names of functions referenced within the tree: $b_1, b_2, \ldots b_n$. We insert `a` into a graph and add edges to each node $b_1, b_2, \ldots b_n$. Edges from `a` to `a` due to recursive calls are removed, and so are repeated function names.

We repeat this procedure for all functions in the program. As we are interested in call chains—i.e., calls to other functions from a called function—we calculate the transitive closure of this graph. As trees are special instances of a directed acyclic graph, this new graph defines the reachability of one function from another, providing an over-approximation of control flow. This can be determined using Floyd Washall in $O(n^3)$, though it could be reduced to $O(E + \mu n)$ using Purdom's algorithm if dealing with large programs.

As implemented, inlining pragmas will be added to any function which lies in the call chain of a profiled hotspot, whether that call site of the function is indeed hot or not. Therefore, some of these call chain branches may not benefit from being coerced with pragmas. However, research has suggested that some extraneous inlining may not affect performance very much [45]. Thus, our experiments err on the side of over-inlining.

6.3.3 Pragma Placement

Our analysis produces a recommendation of which function names should receive an inlining pragma in each file. It is possible to annotate the source files themselves with these pragmas; however, we pass the recommendations directly to GHC with the implementation described in Section 6.4. With either approach, functions are inlined at their definition site, as opposed to individual call sites.

6.4 Implementation

To produce inlining recommendations, we compile packages with GHC with profiling enabled to produce `.prof` files and module ASTs. We then pass the profiling information and ASTs to a parsing script that collects the names of functions associated with the functions identified in the profiles' hotspots and output them into JSON objects. We only identify functions declared at the top level of the module and not locally scoped functions which receive new names in the Renamer pass to avoid name collision with other locally scoped functions in the same module. Packages contained 4 to 609 such function names with an average of 130 for each package.

We then binarize the JSON into data structures which can be passed to a modified version of GHC which can then place pragmas in the binaries' indicated functions. A diagram depicts the production of the binaries in Figure 6.2. It is not necessary to have a version of GHC that is modified to work with these binaries: the pragma recommendations can be added manually, but binaries made for faster experimentation and had the combined benefit of being overridden by developer pragmas as described in Section 6.6.3, resulting in even greater speedups.

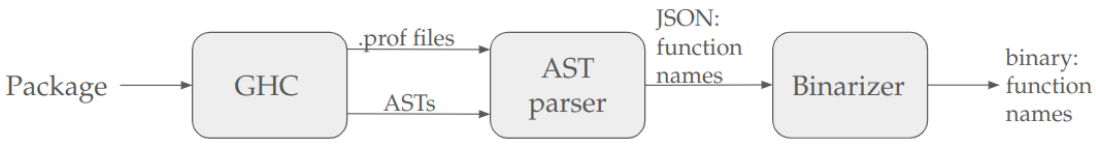


Figure 6.2: The process of retrieving recommended function names for inlining in a binary format.

6.4.1 Collection of Profiling Information

We first build each package, enabling tests and profiling, with the command:

```
cabal new-build all --enable-tests
--enable-profiling --with-compiler=<path>
```

We then generate `.prof` files which contain hotspots for each of the test and benchmark targets specified in each project's cabal file, by passing profiling flags to `cabal run`:

```
    Mon Feb 19 14:00 2024 Time and Allocation Profiling Report  (Final)

       Main +RTS -p -RTS

    total time  =        0.25 secs   (246 ticks @ 1000 ms, 1 processor)
    total alloc = 705,165,992 bytes  (excludes profiling overheads)

COST CENTRE MODULE  %time %alloc


COST
CENTRE   MODULE    SRC                              %time %alloc

funcA    DA.M1     src/DA/M1.hs:<loc>                63.4   78.5
funcB    DB.M2     src/DB/M2.hs:<loc>                 9.3    4.0
funcC    DA.M1     src/DA/M1.hs:<loc>                 6.9    4.7
funcD    Test1     test/Test1.hs:<loc>                3.3    0.9
funcE    Test1     test/Test1.hs:<loc>                3.3    2.6
funcF    Test2     test/Test2.hs:<loc>                2.4    0.7
funcG    DB.M3     src/DB/M3.hs:<loc>                 1.6    0.4
funcH    DB.M3     src/DB/M3.hs:<loc>                 1.2    2.5
```

Figure 6.3: A fabricated example profile report showing significant cost centres along with their module names, source code locations, and percent of total run time and allocations.

```
cabal run <test> --with-compiler=<path> -- +RTS -p
```

Significant hotspots for each target are recorded in each `.prof` file in a table format similar to the fabricated example below in Figure 6.3. More information on profiling is given in Section 2.4.3.

Builds with profiling enabled are used to collect profiling information only and never used for timings in experiments because profiling instrumentation affects GHC's optimizations and therefore also execution times.

6.4.2 Coerced Inlining of Hot Call Chains and All Related Functions in GHC

Following the parsing of ASTs and collection of functions associated with profiling hotspots, as described in Section 6.3.2, a Haskell program then reads in the JSON containing hot call-chain information for each package and serializes it into a binary object. GHC takes in the specific binary for each package and compiles that package again, applying an inlining pragma to any function identified as related to a hotspot for each module.

For this study, every function name in a binary receives the same type of pragma, `INLINE` or `INLINABLE`, for each experimental compilation of a package.

6.5 Experimental Setup

Experimental design details for benchmark package selection, response measurement, and determination of inlining policies explored are explained as follows.

6.5.1 Data Collection

Selection of Packages

We select 17 real-world packages from Stackage [55], a collection of repositories containing Haskell packages grouped into Nightly Builds, where Nightly Builds are sets of packages which will build and pass all of their tests together. These packages come from the Stackage Nightly build[1] mentioned in Hollenbeck et al. [77]. We base package selection on the following criteria:

> The packages have to successfully build and run tests and benchmarks (if present).
>
> The packages' tests and benchmarks (if present) have to run for more than 0.5 seconds.
>
> The tool must successfully parse dependencies from the packages' ASTs emitted at compile time. This may fail in the event that none of a package's hotspots occurred in the `src` folder. For example, the only substantial hotspots for some packages may have occurred in their test code. Not all packages in Stackage are structured to have a `src` folder, but this is one easy way to reduce the likelihood that speedups are observed from inlining code in test files.

We only explore hotspots in the `src` folders, with the intent to optimize the packages' core functionality and the likelihood that any observed speedups will generalize to other tests and benchmarks. In actuality, some packages do not contain all of their core functionality in a `src` folder. Figure 6.4 shows a distribution of the percent of significant cost centers located in the `src` folder as opposed to other folders, on a per-executable basis. About 75% of the executables have at least 25% of their hotspots residing in the `src` folder. Note: five executables in Figure 6.4 indicate 0 hotspots in the `src` folder; however, other executables in those packages contain at least one hotspot in the `src` folder.

Notably, six of the packages used for the experiments in Chapter 4 fit the above selection criteria and were included, as explained further in Section 6.6.7. The packages `hw-rankselect`, `nonempty-containers`, `ListLike`, and `metrics` were excluded.

---

1. stackage-nightly-2020-01-31

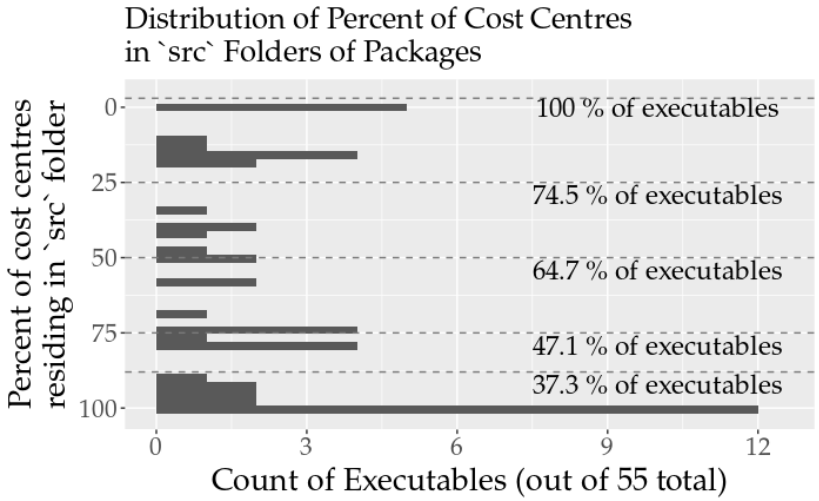


Figure 6.4: A rotated histogram of the occurrence of executables' hotspots in packages' `src` folders, as found in profiling reports. Executables include individual tests and benchmarks indicated in packages' `cabal` files.

Rewrite Rules

We exclude packages with rewrite rules from the study because of the additional complexity that would be required to accommodate them. A rewrite rule replaces pattern-matched code with another indicated expression and may be accompanied by an `INLINE` pragma to allow the substitution to happen before optimization. Rewrite rules may be specified in a `RULES` pragma near the declaration of the function they rewrite. This example comes from the GHC user guide [161]:

```
{-# RULES
"map/map"  forall f g xs.  map f (map g xs) =
              map (f.g) xs
  #-}
```

The rewrite rules pragma replaces the expression given on the left-hand side of the rule with the right-hand side of the rule, with appropriate substitution. A `RULES` pragma may be accompanied by an `INLINE` or `NOINLINE` pragma to ensure that the compiler does not optimize the expression before it can be matched to a rule; however, adding an inlining pragma is not a requirement when writing rules.

Randomly Generated Tests

Because random test generation can lead to inconsistent test and benchmark times, we perform the same modification to a local copy of QuickCheck as seen in Hollenbeck et al. [77], setting the random seed to make test generation consistent, and use that copy to replace any of the packages' QuickCheck dependencies.

6.5.2 Measurement of Performance Change

Experiments were run on an Intel Core i7-8700K CPU with 12 cores and a 3.7 GHz processor with frequency at 1.5 GHz and boost disabled to control overclocking for reproducible timings.

Performance improvement was measured in terms of real execution time against a baseline of the Cabal `build` command's default GHC optimization level of O2, collecting 10 executions of each package's code with `INLINE` and `INLINABLE` pragmas removed. Following experimental changes, 10 timings were again collected for each package. The minima for the baseline were compared against the minima for the changes to determine speedups. All compilation time, including for tests and benchmarks, was excluded in collected data by discarding the first test run timing for any package.

6.5.3 Inlining Policies

We name and describe all of the inlining policies for the experiments in this list.

Naive:
Remove the developers' pragmas and insert `INLINABLE` pragmas on only hot functions.

HCC `INLINE`:
Remove the developers' pragmas and insert `INLINE` pragmas along hot call chains.

HCC `INLINABLE` :
Remove the developers' pragmas and insert `INLINABLE` pragmas along hot call chains.

Best-case HCC `INLINABLE` **or** `INLINE`:
Remove the developers' pragmas and, per package, choose the better of HCC `INLINE` or HCC `INLINABLE`.

Pragmas and HCC `INLINABLE`:
Keep the developers' pragmas and insert `INLINABLE` pragmas along hot call chains.

Pragmas and HCC `INLINE`:
Keep the developers' pragmas and insert `INLINE` pragmas along hot call chains.

Pragmas and Best-case HCC `INLINABLE` **or HCC** `INLINE`:
Keep the developers' pragmas and, per package, choose the better of HCC `INLINE` or HCC `INLINABLE`.

For the experiments choosing a best-case policy, we calculate the results of both candidate policies for each package and manually select the policy with faster run time for that individual package.

6.6 Results

Table 6.2 provides the mean speedups of the 7 policies described in Section 6.5.3, and more explanation of each experiment follows in this section.

Method	**Speedup**
Naive	2%
HCC `INLINE`	4%
HCC `INLINABLE`	7%
Best-case HCC `INLINABLE` or `INLINE`	9%
Pragmas and HCC `INLINABLE`	8%
Pragmas and HCC `INLINE`	9%
Pragmas and Best-case HCC `INLINABLE` or HCC `INLINE`	10%

Table 6.2: Summary of geometric mean speedups along various policies of adding inlining pragmas via profile guidance by identifying hot call chains (HCCs).

6.6.1 The Naive Approach

An immediate question for a profiling approach would be: What improvement can we see if we simply add inlining pragmas to the function names of hotspots identified in the profiling report? In our packages, such an approach gives a geometric mean speedup of about 2%. Table 6.3 shows that there are only 5 packages in the Naive experiment which have a speedup over 1%.

Notably, two of these packages have a significant speedup of 25% and 17%—midi-0.2.2.2 and set-cover-0.1, respectively. However, those same packages have maximum observed speedups of 51% and 23% when inlining along the hot call chains.

Package Name	Hotspots	Naive Speedup
css-text-0.1.3.0	4	1%
monoid-subclasses-1.0.1	4	1%
texmath-0.12	2	1%
midi-0.2.2.2	20	25%
set-cover-0.1	17	17%

Table 6.3: Packages with a speedup of 1% or more using a naive profile-guided inlining method.

6.6.2 Hot Call Chains Without Developer Pragmas

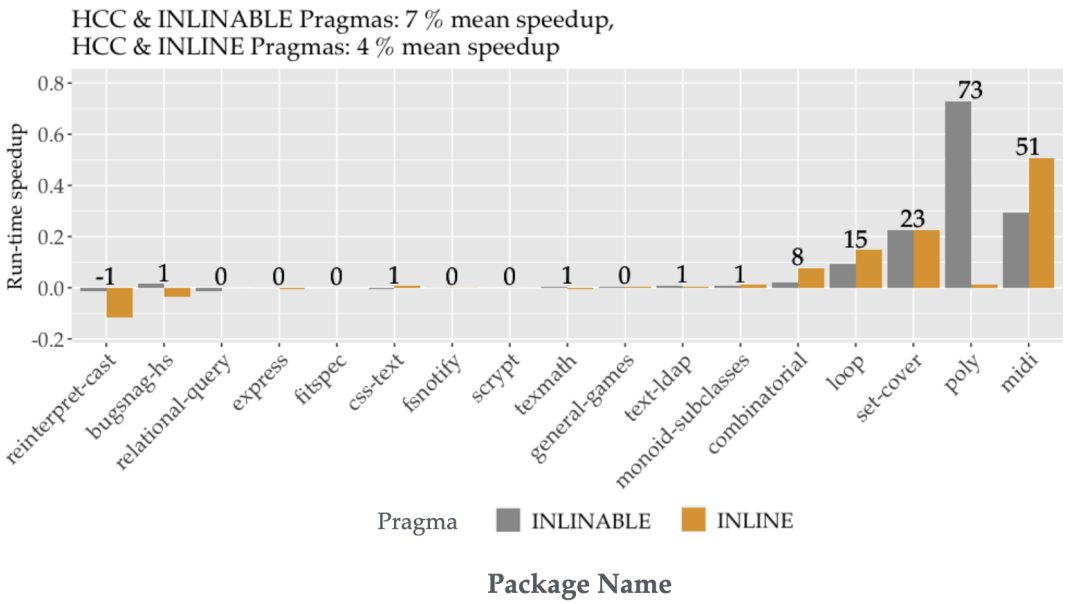


Figure 6.5: Inserting `INLINABLE` pragmas along hot call chains versus inserting `INLINE` pragmas along hot call chains. Both timings compare against a baseline of timings with all inlining pragmas removed. The numbers above each pair of bars represents the highest percent of the two speedups observed for each package.

Figure 6.5 shows a comparison of speedups over the collection of packages when placing either `INLINE` or `INLINABLE` pragmas along hot call chains. The following two sections discuss these results.

HCC `INLINE`

The `INLINE` pragma along hot call chains gives much better run times for `midi`, `loop`, and `combinatorial`; however, it also introduces significant slowdowns in the two packages `reinterpret-cast` and `bugsnag`, likely due to the caveats of the `INLINE` pragma mentioned in Section 2.2.5—that is, the `INLINE` pragma may push GHC to inline items which are so large that they degrade performance. Additionally, `INLINE` fails to produce any substantial speedup in `poly`, which shows the highest overall speedup of 73% with the `INLINABLE` pragma on hot call chains. Overall, the HCC `INLINE` policy gives a geometric mean speedup of 4%.

HCC `INLINABLE`

The HCC `INLINABLE` policy gives a geometric mean speedup of 7%. As mentioned, Section 2.2.5 explained GHC may fail to see the net benefit of a large inlining with a good potential speedup with only an `INLINABLE` pragma; however, an `INLINE` pragma may result in a large inlining with a poor performance result. Using `INLINE` on call chains may produce both a number of higher speedups on some packages and a number of slowdowns on others, whereas the more conservative `INLINABLE` is more likely to avoid instances of massive slowdown caused by aggressive inlining, according to this dataset.

Best-case HCC `INLINABLE` or HCC `INLINE`

For each package, we then compare the policies of placing all-`INLINE` or all-`INLINABLE` pragmas across the entire hot call chain. Choosing the best-performing policy for each package brings the mean speedup to 9%.

To further illustrate, Table 6.4 shows, per package, which policy performed better and what its observed speedup was. In total, the HCC `INLINE` policy was better in 6 packages, the HCC `INLINABLE` policy was better in 5 packages, and they were about the same in another 6 packages.

6.6.3 Inlining Hot Call Chains with Developer Pragmas

Do the developer pragmas in the packages effect a speedup? To address this, we compare execution times of packages with developer pragmas against execution times of packages with their developer pragmas removed. The developer pragmas achieve a mean speedup of 4% over no pragmas. The DEV bars in Figure 6.6 show developer pragma speedups for each package.

As seen in Figure 6.7, if we inline along hot call chains *and leave in* developer pragmas, we see about a 4% mean speedup over the timings of the packages run with just the developers' inlining pragmas versus without.

Package	Better Policy	Better Policy Speedup
reinterpret-cast	INLINABLE	-1%
bugsnag-hs	INLINABLE	1%
relational-query	INLINE	0%
express	EITHER	0%
fitspec	EITHER	0%
css-text	INLINE	1%
fsnotify	EITHER	0%
scrypt	EITHER	0%
texmath	INLINABLE	1%
general-games	EITHER	0%
text-ldap	INLINABLE	1%
monoid-subclasses	EITHER	1%
combinatorial	INLINE	8%
loop	INLINE	15%
set-cover	INLINE	23%
poly	INLINABLE	73%
midi	INLINE	51%

Table 6.4: A breakdown, per package, of which hot call-chain policy performed better and what speedup was observed from it. In cases where the recorded speedups are the same, the policy "EITHER" is listed. Speedups were rounded to the nearest percent, and the geometric mean of all best-case speedups is 9%

It is important to note that, in our implementation using a binary, developer pragmas will override any pragma recommendations given along hot call chains. When compiled through the system described in Figure 6.2, call chain pragmas will only be attached to functions that have no pragmas attached to them already.

As seen in Figure 6.6, we compile the packages with just their developers' inlining pragmas (DEV), developers' pragmas and hot call chains with `INLINABLE` pragmas (INLINABLE), and developers' pragmas and hot call chains with `INLINE` pragmas (INLINE). We compare these timings of each package to a baseline of the package with all inlining pragmas removed.

The geometric mean speedups for experiments with the hot call-chain technique applied to packages *while leaving developer pragmas in* are as follows:

Pragmas and HCC `INLINABLE`**.** Using `INLINABLE` along hot call chains with developer pragmas gives about an 8% mean speedup, as depicted in Figure 6.6.

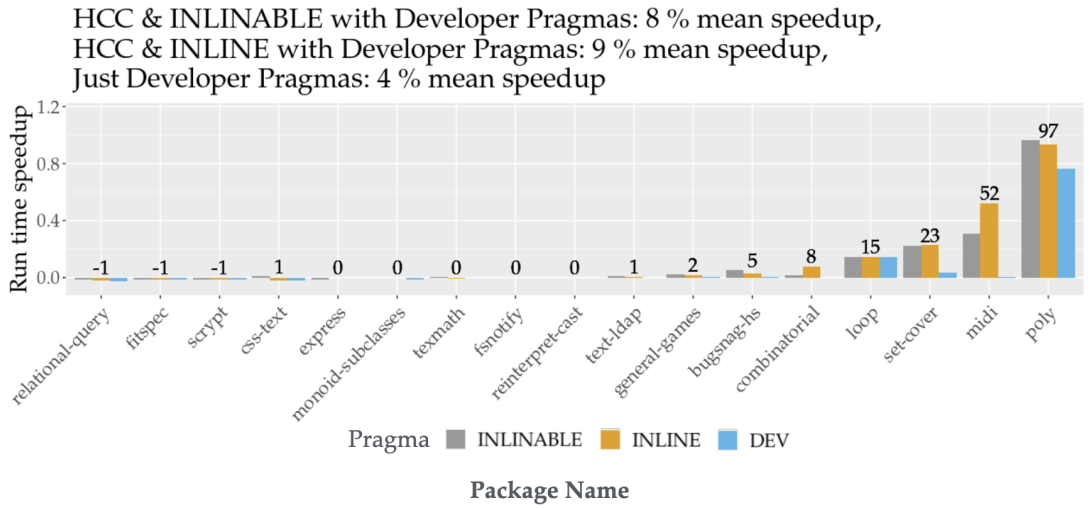


Figure 6.6: Leaving developer pragmas in and combining with hot call chain `INLINE` pragmas (INLINE); leaving developer pragmas in and combining with hot call chain `INLINABLE` pragmas (INLINABLE); and developer pragmas alone (DEV). Speedups are comparisons against default GHC with all inlining pragmas removed from packages.

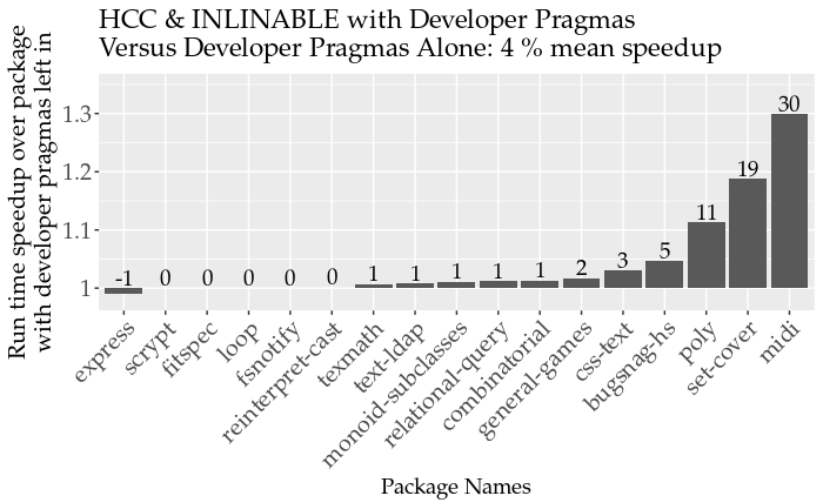


Figure 6.7: Leaving developer pragmas in and compiling with the additional `INLINABLE` pragma recommendations along hot call chains. Speedups are over package timings with only developer pragmas.

Pragmas and HCC `INLINE`. Combining hot call chain `INLINE` pragmas with developer pragmas gives about a 9% mean speedup, also depicted in Figure 6.6.

Pragmas and Best-case HCC `INLINABLE` or `INLINE`. Then taking the better policy of all `INLINE` or all `INLINABLE` on hot call chains for each package, along with developer pragmas, gives about a 10% mean speedup.

Prioritizing Developer Pragmas Along Hot Call Chains

When implemented with a binary which passes in function names to receive a pragma, as opposed to placing pragmas directly in source code, this system has an added advantage of prioritizing the package developers' choice of `INLINE` or `INLINABLE` at each function declaration.

When inlining along hot call chains, prioritizing the pragma choice of the developer helps to avoid the potential performance pitfalls of using the wrong pragma with a uniform approach of one or the other. Thus, combining call chain inlining with well-placed pragmas already present in the source code resulted in better speedups than either method alone.

6.6.4 Comparing the Number of Pragmas: Hot Call Chains Versus Developers'

How do the number of pragmas inserted via the hot call chain method compare to the number of existing pragmas in packages?

Package	Dev. Pragmas	HCC Pragmas	Dev. Speedup	HCC Speedup
poly	99	118	77%	73%
set-cover	17	46	3%	23%
loop	8	8	15%	15%
midi	2	183	1%	51%

Table 6.5: The speedups and number of pragmas for hot call chains versus those included by developers. HCC with `poly` and `set-cover` uses all `INLINABLE`, and HCC with `loop` and `midi` uses all `INLINE` pragmas. Developers use either pragma on any function at their discretion.

Table 6.5 shows the number of pragmas present in the four packages exhibiting the greatest speedup with the hot call chain method, comparing the number of pragmas placed by developers versus the number of inlining pragmas recommended along hot call chains. For `loop-0.3.0`, our method predicts the same number and location of inlining pragmas as the developers, but these 8 functions are also the only candidates for pragmas by our criteria. The developers for `poly` achieve a 77% speedup through their choice of 99 `INLINE` and `INLINABLE` pragmas, whereas our hot call chain method with 118 pragmas—all `INLINABLE`—comes within about 4% of that speedup at 73%.

For `set-cover`, HCC recommends 46 pragmas versus the developers' 17, but HCC achieves a 23% speedup versus the developers' 3%. The package midi-0.2.2.2 only has 2 developer pragmas, which give it a speedup that may fall within the range of noise at 0.5%; however, our method recommends 183 `INLINE` pragmas which yield a 51% speedup.

6.6.5 Adjusting the Threshold of Hot Call Chain Inlining

We set a higher threshold on the selection of hotspots over GHC's default profiling hotspot selection to see if more conservative inlining would produce a similar speedup. We filter for only hotspots with 1% or more of run time, no longer considering allocations, and apply `INLINABLE` pragmas along hot call chains. Figure 6.8 compares this experiment to our previous results of hot call chains with `INLINABLE` along the GHC profiler's default significant cost centres.

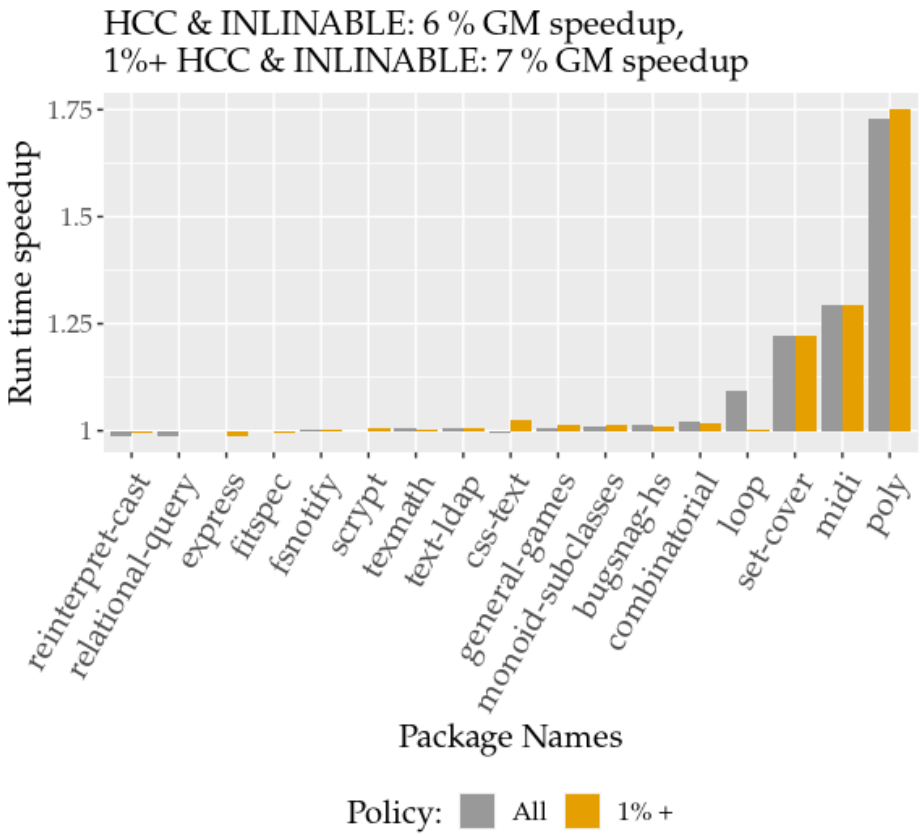


Figure 6.8: Coercing inlining for hot call chains for hotspots which take 1% or more of only run time versus hot call chains for GHC's default time and allocation hotspots.

Three packages with the largest speedups—poly-0.3.3.0, midi-0.2.2.2, and set-cover-0.1—achieve about the same performance gains as seen when simply applying `INLINABLE` pragmas to all default hotspots in the `src` folder over these particular packages. Notably, the package `loop-0.3.0` does not receive a speedup. This package has a Hackage description of "Fast loops (for when GHC can't optimize forM_)". Its `src` folder has 8 top-level function declarations, and the developers assigned `INLINE` pragmas to all 8 of them. Two of these functions did not meet the 1%+ run time threshold.

Results were otherwise very similar across packages, and minor differences per package may be due to noise in measurements and rounding.

6.6.6 Effect On Binary Size and Compilation Time

Table 6.6 shows that the mean compile time increase for the 17 packages when applying the `INLINE` pragma along hot call chains is 16%; and for the `INLINABLE` pragma, it is 3%. For comparison, we applied `INLINE` and `INLINABLE` pragmas to *all* top-level functions identified in the ASTs and saw an 18% and 4% increase in compile time, respectively.

Policy	Compile Time Mean Increase	Package Mean Size Increase
`INLINE` All	18%	20%
`INLINABLE` All	4%	11%
HCC `INLINE`	16%	6%
HCC `INLINABLE`	3%	1%

Table 6.6: Increases in mean compilation time and package sizes using four inlining policies. Numbers are rounded to the nearest percent.

However, the size of the compiled packages, as also indicated in Table 6.6, were smaller with hot call chains than with the pragma-on-everything approach. For `INLINE` and `INLINABLE` on all, we see a 20% and 11% increase in code size. For the hot call chains, in comparison, we see for HCC `INLINE` and HCC `INLINABLE` a 6% and 1% increase in size, respectively.

As mentioned in Section 6.4, we only collect function names that are declared at the top level of the module for any of these experiments. This may contribute to the modest change in compilation time and size in comparison to the baseline package size with no pragmas.

Table 6.7 displays the build sizes for the packages for both HCC `INLINABLE` with default GHC profiling hotspot thresholds and when filtering for only run time greater than 1% per Section 6.6.5. For the default profiling thresholds, all but two of the packages have less than 1% code size increase: `midi` and `poly`, at about a 7% and 6% increase, respectively. The 1%+ threshold had a mean size change of -0.4%. Notably, `text-ldap` became significantly smaller at -19.36% but had no time speedup, and `midi` still increased by 7%.

6.6.7 Comparison Against Magic Numbers Alteration

In Chapter 4, we did an extensive search of a compilation space of 12 magic numbers hand-encoded in GHC's inliner, where the magic numbers represented either size estimates or discounts to determine inlining decisions for various program expressions in Core IR. The study presents the most recent related work for GHC's inliner and presents inlining experiments over a benchmark of 10 packages chosen from Stackage to demonstrate room for improvement for GHC's inlining heuristics. We use the same Stackage Nightly build in this

Package	Size MB	HCC All	Size %Inc.	HCC 1%	Size %Inc.
bugsnag-hs	46.37	46.38	0.02	46.38	0.02
combinatorial	20.77	20.83	0.29	20.82	0.24
css-text	40.91	41.03	0.29	41.03	0.29
express	104.13	104.49	0.35	104.48	0.34
fitspec	27.84	27.91	0.25	27.87	0.11
fsnotify	15.89	15.90	0.06	15.90	0.06
general-games	42.91	43.09	0.42	43.08	0.40
loop	42.24	42.23	-0.02	42.00	-0.57
midi	45.22	48.32	6.86	48.31	6.83
monoid-subclasses	57.34	57.37	0.05	57.36	0.03
poly	53.75	57.08	6.20	57.07	6.18
reinterpret-cast	42.04	42.13	0.21	42.12	0.19
relational-query	53.13	53.14	0.02	53.14	0.02
scrypt	24.34	24.35	0.04	24.34	0
set-cover	30.45	30.61	0.53	30.61	0.53
texmath	173.32	173.34	0.01	173.33	0.01
text-ldap	29.86	30.03	0.57	24.08	-19.36
Mean			0.9%		-0.4%

Table 6.7: Package sizes with developer pragmas removed (Size MB); compiled sizes with inlining pragmas added to all reported active hotspots' call chains (HCC All); sizes with inlining pragmas added to all reported hotspots with more than 1% run time reported (HCC 1%); and percent increase in size for the two policies (Size %Inc.).

study to measure improvement against recent work. In Chapter 4, we found a speedup of a single best configuration of magic numbers which yielded a 7% speedup for 10 packages when the developer pragmas were removed versus GHC's default magic number values. Some additional speedups were reported for related experiments.

The best-case magic number configuration from Chapter 4 only produced a 1.3% speedup across the original 10 packages depicted in Table 4.2 on the architecture we used for this set of experiments, but Figure 6.9 shows we observed speedups above 3% for midi-0.2.2.2, set-cover-0.1, and `reinterpret-cast-0.1.0`. In comparison, hot call chains with `INLINABLE` showed a 12% speedup across those 10 packages. If we choose the best of HCC `INLINE` or `INLINABLE` across the 10 packages with pragmas removed, we get a 14% speedup.

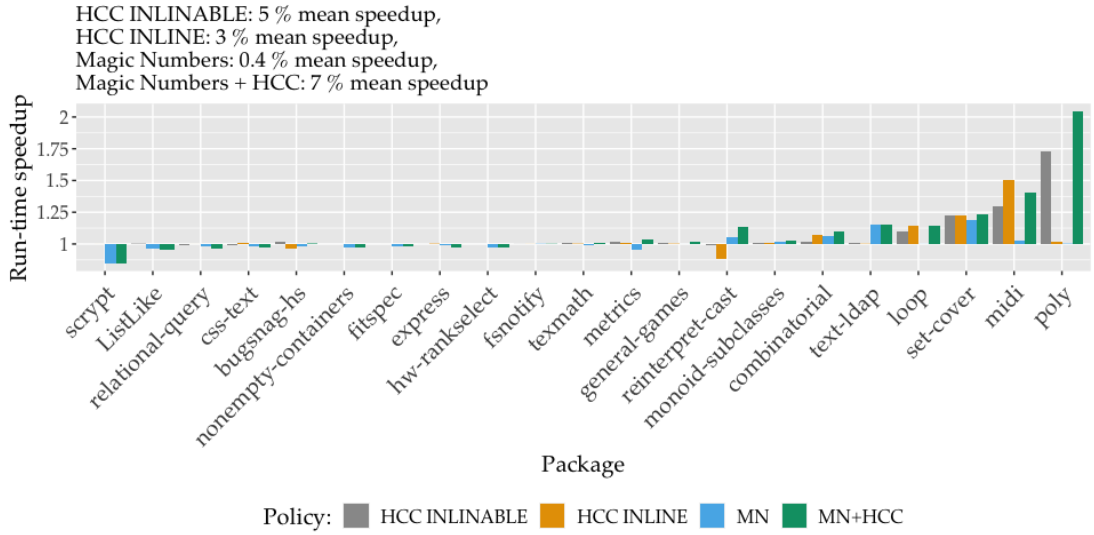


Figure 6.9: Speedups of HCC with `INLINABLE` pragmas, speedups of HCC with `INLINE` pragmas, speedups of packages compiled with the best configuration of magic numbers (MN) for packages without inlining pragmas (configuration 229), and speedups with MN 229 and HCC `INLINABLE`. The baseline is GHC with default magic numbers and inlining pragmas removed.

Across all 17 packages in our study, we saw about a 0.4% mean speedup with the magic numbers. Like the other experiments, the baseline was GHC with its default magic numbers and inlining pragmas for all packages removed.

Figure 6.9 additionally shows the speedups of hot call chain inlining per package with either `INLINABLE` or `INLINE` pragmas along all hot chains versus the speedup for the best configuration of magic numbers without pragmas (MN). The altered magic numbers seem to additionally show speedups for `text-ldap-0.1.1.13` and `combinatorial`. For the remaining packages outside of the study, however, there is frequently little effect or a small but significant negative effect. Of the 21 total packages timed with the magic numbers, 11 of them showed a slow-down of 1% or more with just magic number alteration (MN). For hot call chains, either choice of pragma only gave 2 packages a slow-down of 1% or more.

It should be noted that `hw-rankselect`, `nonempty-containers`, `ListLike`, and `metrics` were excluded from the packages for hot call-chain experimentation because they did not meet the selection criteria; however, we included their results in Figure 6.9 for comparison against our work in Chapter 4.

6.6.8 Changing Input Data

The packages contained a smaller number of executables marked as benchmarks, in comparison to test suites. Compiling the packages with `INLINABLE` along hot call chains and timing these benchmarks gave similar results to those seen in the test suites. Figure 6.10 shows that poly-0.3.3.0 and set-cover-0.1 had speedups of 34% and 25%, respectively. The remaining three packages with benchmarks received no speedup. Notably, `loop-0.3.0` also received no speedup despite its tests achieving a 15% speedup with hot call chain inlining; however, running `loop-0.3.0` with the developer pragmas on the benchmarks versus without also did not achieve the 15% speedup seen with the test suites.

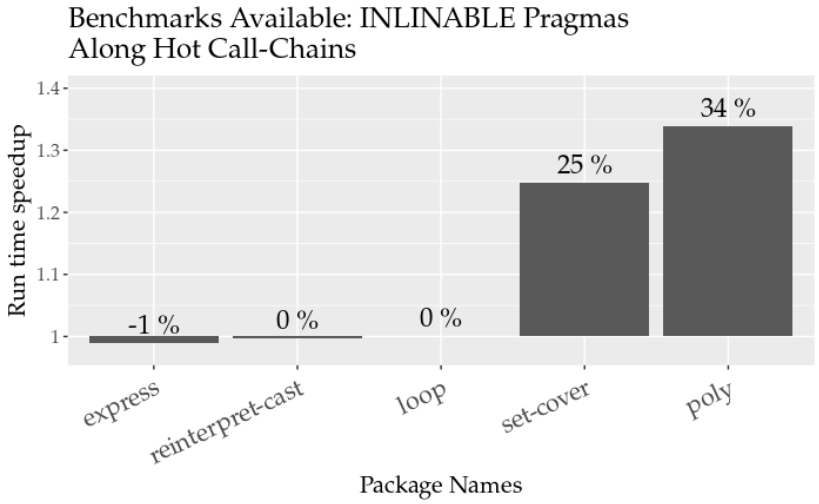


Figure 6.10: Inserting `INLINABLE` pragmas along hot call chains in the benchmarks available in the dataset versus no pragmas.

6.7 Conclusions

The hot call chain technique uses profile information to identify hotspots in packages, then performs a simple lexical analysis of the AST to find functions related to those hotspots. GHC is instructed to inline hot call chains with pragmas, which produces a significant speedup in compiled programs. We have shown this technique is more effective than inlining hotspots on their own. In the packages upon which we demonstrate this technique, there exists a subset of packages for which the method produces a very large speedup; and for the remaining packages, performance and build sizes remain virtually the same. The computational complexity of the hot call chain approach is worst-case quadratic in practice, and combining it with manually well-placed `INLINE` or `INLINABLE` pragmas often boosts performance, as shown by using the technique with pre-existing developer pragmas in the benchmark packages. Future work could determine when either pragma would be preferable for specific functions.

6.8 Summary

This chapter presented a conceptually simple technique that over-estimates call chains connected to functions determined "hot" by profiling with the Glasgow Haskell Compiler, then places inlining pragmas (either `INLINE` or `INLINABLE`) along each function definition within those chains. The empirical data presented demonstrated that this over-estimation produces a speedup in a subset of packages, which combined with developer pragmas produces a geometric mean speedup of 10% with the better choice of inlining pragma, while leaving the binary size and run time of the other packages unaffected.

The technique in this chapter follows upon insights gained through investigatory work done in Chapter 4 and Chapter 5 regarding information pertaining to developer pragmas, the reduced complexity of inlining at the source code rather than during Simplification in the middle of the compiler, and the insight to inline along entire call chains rather than at individual function declarations.

Chapter 7

Conclusions

This thesis provided an exploration of the challenges of optimizing inlining in the Glasgow Haskell Compiler, with an emphasis on machine learning approaches and the use of a simple static analysis. Chapter 4 established room for improvement over the long-standing default heuristics through experimentation and empirical evidence, also providing tooling to aid in the supervised learning discussed in Chapter 5. It provided an empirical evaluation of inlining strategies through a search of the heuristic space created by varying the magic numbers in GHC, presenting a set of magic number configurations to yield speedups across a set of real-world Haskell packages and additionally a simple clustering method to assign packages to these configurations.

Chapter 5 introduced the machine learning approaches attempted to optimize GHC's inliner and discussed why these attempts failed. Features pertaining to the syntax of individual inlining decisions seemed to have inadequate information to reach the potential speedups exhibited in Chapter 4, motivating the case to approach the problem on a level more holistic to the entire program, as presented in Chapter 6.

In Chapter 6, a simple flow analysis obtained by the abstract syntax tree yielded substantial speedups, illustrating the importance of control flow to optimize inlining in Haskell. Although previous work suggested the potential for control flow to effectively guide inlining decisions in functional languages, the problem of determining control flow remained prohibitively difficult to effect its use for inlining. Hot call-chains, as presented in Chapter 6, demonstrate that using an over-approximation of control flow which errs on the side of inlining more execution paths than those strictly determined to be hot by profiling still produces a significant speedup over GHC's default inlining heuristic.

We summarize the salient contributions of the thesis in this chapter and further critically analyze the work done therein, concluding with directions for future work.

7.1 Summary of Contributions

7.1.1 A Benchmark Framework for GHC

Section 4.2.3 established that Haskell has no formal benchmark suite that is up-to-date and fully adequate for optimizing GHC, and Section 4.2.3 described a framework which allows real-world packages from Hackage to be used as benchmarks. As Hackage is a live collection of continually updated packages of real-world code submitted by the Haskell community, appropriate sampling from Hackage for benchmarking allows for the creation of representative and relevant benchmarks.

7.1.2 An Empirical Investigation of GHC's Inlining Decisions

Section 4.5 provided an analysis of inlining decisions in GHC, including a characterization of observed "good" decisions according to the analysis. Such an analysis can provide the compiler's developers with insight into what changes may potentially be made to the inliner in order to encourage more of such "good" decisions to be made.

Analysis in Section 4.5 demonstrated that every package in the work performed in Chapter 4 prefers its own unique set of parameterized magic numbers. This observation suggests the inadequacy of program features at the point of Simplification to fully guide inlining decisions in GHC, which is a large part of GHC's current inlining approach.

These analyses additionally demonstrated, with empirical evidence, how automated tuning techniques may be used to improve performance of inlining within GHC.

7.1.3 A Simple Cluster-Based Predictive Model for Performance Improvement

Section 4.6 introduced a simple predictive model, based upon clustering and observation of run-time speedups, to assign packages to the set of "best" magic number configurations given in Section 4.5.1.

7.2 Observations from Experiments to Improve GHC's Inlining with Machine Learning

Chapter 5 presented three machine learning approaches to improve GHC's inliner: a genetic algorithm, neural networks, and graph neural networks. Analysis of these experiments yielded some key observations about the challenges of inlining in GHC, which laid the foundations for the approach in Chapter 6.

In Section 5.3, neural networks within the middle of the compiler were trained to predict inlining decisions at the point of Simplification of CoreIR to CoreIR. At this point, the compiler can potentially make millions of inlining decisions. Training data was generated by setting GHC's magic numbers to those preferred most by each package mentioned in Chapter 4 and recording all `(features, decision)` outcomes. Although the models were trained to high accuracy, they could not achieve the same compile times seen in Chapter 4. It was observed that there was a tiny fraction of training data which had identical features but also had both 0 and 1 labels. If the ambiguity is assumed to be caused by an inline pragma, Section 5.4.8 showed by experimentation that erring on the side of the pragma can lead to over-inlining.

These neural networks were intended to be used as a seed population for a genetic algorithm, as experiments in Section 5.2 which trained new inliner populations from scratch by using a genetic algorithm were demonstrated to compile, run, and evolve prohibitively slowly.

Section 5.4 trained graph neural networks to approach the problem instead at the source-code level, to predictively place inlining pragmas at function declarations. Training data was produced by adding or removing individual pragmas and noting run-time differences. Despite high training accuracy again, no significant speedups were observed—even when overfitting and predicting on the same package. Modifying the data collection strategy from adding pragmas one by one to removing pragmas one by one and comparing results demonstrated that inlining decisions are highly dependent upon each other.

Manual exploration of the code revealed that developer pragmas with the highest impact on performance in the packages were related to each other via chains of connected control flow, and the functions at the top of these chains showed up as bottlenecks in a profiling report. These insights, combined with those listed above, lead to the approach presented in Chapter 6.

7.2.1 A Simple Approximate Hot Call-Chain Algorithm for Inlining Decisions in GHC

Drawing from the observations in Chapter 5, Chapter 6 presented a simple technique, hot call-chain inlining, to predict placement of inlining pragmas in Haskell source code based upon profiling information and an over-approximated control flow graph derived from a program's abstract syntax tree. This technique is worst-case quadratic in practice and easier to conceptualize than a more precise control flow algorithm. It may additionally be implemented to defer to inlining pragmas already included in package code, where allowing the human judgment of `INLINE` or `INLINABLE` in locations of existing pragmas is consistently shown to produce the biggest speedup when combined with hot call-chains.

The use of hot call-chains on a set of 17 real-world Haskell packages produced with the framework described in Section 4.2.3 demonstrated a 10% mean speedup when combined with a choice of `INLINE` or `INLINABLE` pragmas along the hot call-chains of each package with the developers' inlining pragmas left undisturbed in the source code. Without developer pragmas but allowing the better choice of all `INLINE` or all `INLINABLE` for each package, the hot call-chain approach gives a 9% mean speedup.

Furthermore, the hot call-chain technique produces only a 1% mean increase in code size across all packages, with no packages seeing a size increase above 7%. No significant runtime slowdowns manifested in the dataset with this technique.

7.3 Critical Analysis

This section provides a self-evaluation of some of the work presented in this thesis in a critical manner, addressing some potential concerns which may arise regarding design decisions and execution.

7.3.1 Selection Bias

The packages chosen for the experiments in Chapters 4 and 6 all contain inlining pragmas, as inserted by the packages' developers. The intention behind this decision was that some speedup may have already been discovered by the developers and may contribute toward an oracle indicating a best-case inlining strategy against which solutions may be compared. In actuality, developer pragmas did not consistently give speedups, and it is possible that code sub-selected from Hackage which contains inlining pragmas may differ from code which is otherwise randomly sampled from Hackage. However, inclusion of the packages in Chapter 4 was necessary in the experiments of Chapter 6 as a basis for comparison, as no other directly related recent work had been done in the area of inlining in the Glasgow Haskell Compiler.

7.3.2 Exclusion of Local Functions from Hot Call-Chains

The experiments in this thesis used GHC version 8.10.3. When attaching inlining pragmas to code via binary per the method described in Section 6.4, this version of GHC failed to discard some pragma suggestions on local functions which would have caused non-terminating behavior during compilation. We tested the system with GHC 9.8.2, in which the issue appeared to have been fixed and the packages compiled correctly; however, because we could not do a direct performance comparison with packages compiled by GHC 9.8.2 to those compiled with GHC 8.10.3, we instead opted to omit pragma recommendations in hot call chains on locally scoped functions. To do this, we discarded function names which received a randomly generated suffix after the renamer pass, as the random name generation serves to prevent locally scoped names from colliding with each other.

7.3.3 The Use of More Compute Power

Some of the experiments run in this study could have been run with more compute power or on a larger number of machines, if resources were available. This is particularly true for the genetic algorithm presented in Section 5.2. However, analysis of the other experiments in Chapter 5 demonstrated the significance of placement of functions inside call chains associated with hotspots. Scaling the experiments likely would not have produced a model which would produce performance results like those seen in Chapter 6, as no run-time characteristics were included in the training data.

7.3.4 More Aggressive Inlining in GHC

The numbers presented in Table 6.6 suggest that GHC may be able to afford to perform more aggressive inlining overall. Assuming that can be done with no negative effect, the magnitude of the speedups produced by hot call-chains may be affected.

7.4 Future Work

Further substantial speedup may potentially be gained by determining whether to use an `INLINE` or `INLINABLE` pragma at each individual function declaration in the hot call-chains, as the choice between them has been shown to have significant performance effects when examining hot call-chains when overridden by developer pragmas. This problem may be suitable for machine learning.

Because the hot call chain work in this thesis was inspired by real-world code examples where profiled hot spots notably had chains calling down to computationally expensive functions, it would be useful to investigate whether these computationally expensive functions can be identified. Identifying these “expensive” leaf functions, perhaps additionally determining whether they are likely to be called often, and adding inlining pragmas along their associated call chains may yield a performance improvement without the need for profiling. It would still be necessary to determine flow to find these functions’ callers; however, annotating these expressions in the CoreIR may allow for some discovery of it as the code goes through multiple optimization passes.

It may be possible for developers to identify functions as bottlenecks—via, e.g., a new pragma—and allow GHC to perform aggressive inlining optimization during compilation. The entire associated control flow graph may not be uncovered, but repeated optimization passes may be able to uncover some of the control flow to produce some speedup. For example, Section 6.2.1 showed a case where developers placed pragmas along part of a call chain and uncovered a measurable speedup.

As mentioned in Section 7.3, locally defined functions were excluded from hot call-chains because of a bug in GHC 8.10.3. Implementing further experiments in a newer version of GHC where this issue is fixed may produce different results than those seen in this thesis.

The hot call-chain technique is not necessarily limited to use in Haskell. Every aspect of design and implementation may be adapted to another compiler. It may be informative to apply the technique to other functional languages as well.

```
("Just"
 ("(,,,)"
  ("HsGroup"
   ("NoExtField")
   ("XValBindsLR"
    ("NValBinds"
     [("(,)"
       ("NonRecursive")
       ("Bag(Located (HsBind Name))"
        ([({LINENUMBERSsimple.hs:2:1-29 }
           ("FunBind"
            ("NameSet"
             ([]))
            ({LINENUMBERSsimple.hs:2:1-10 }
             {Name: Main.addexclaim})
            ("MG"
             ("NoExtField")
             ({LINENUMBERSsimple.hs:2:1-29 }
              [({LINENUMBERSsimple.hs:2:1-29 }
                ("Match"
                 ("NoExtField")
                 ("FunRhs"
                  ({LINENUMBERSsimple.hs:2:1-10 }
                   {Name: Main.addexclaim})
                  ("Prefix")
                  ("NoSrcStrict"))
                 [({LINENUMBERSsimple.hs:2:12-15 }
                   ("VarPat"
                    ("NoExtField")
                    ({LINENUMBERSsimple.hs:2:12-15 }
                     {Name: name_agd})))]
                ("GRHSs"
                 ("NoExtField")
                 [({LINENUMBERSsimple.hs:2:17-29 }
                   ("GRHS"
                    ("NoExtField")
                    []
                    ({LINENUMBERSsimple.hs:2:19-29 }
                     ("OpApp"
```

```
40                        {Fixity: infixr 5}
41                        ({LINENUMBERSsimple.hs:2:19-22 }
42                         ("HsVar"
43                          ("NoExtField")
44                          ({LINENUMBERSsimple.hs:2:19-22 }
45                           {Name: name_agd})))
46                        ({LINENUMBERSsimple.hs:2:24-25 }
47                         ("HsVar"
48                          ("NoExtField")
49                          ({LINENUMBERSsimple.hs:2:24-25 }
50                           {Name: GHC.Base.++})))
51                        ({LINENUMBERSsimple.hs:2:27-29 }
52                         ("HsLit"
53                          ("NoExtField")
54                          ("HsString"
55                           ("SourceText"
56                            "\"!\"")
57                           {"FastString:" "FASTSTRING"})))))))]
58                  ({LINENUMBERS<no location info> }
59                   ("EmptyLocalBinds"
60                    ("NoExtField"))))))])
61              ("FromSource"))
62             ("WpHole")
63             []))])))
64      ,("(,)"
65        ("NonRecursive")
66        ("Bag(Located (HsBind Name))"
67         ([({LINENUMBERSsimple.hs:(4,1)-(7,41) }
68            ("FunBind"
69             ("NameSet"
70              ([{Name: Main.addexclaim}]))
71             ({LINENUMBERSsimple.hs:4:1-4 }
72              {Name: Main.main})
73             ("MG"
74              ("NoExtField")
75              ({LINENUMBERSsimple.hs:(4,1)-(7,41) }
76               [({LINENUMBERSsimple.hs:(4,1)-(7,41) }
77                 ("Match"
78                  ("NoExtField")
```

```
79                  ("FunRhs"
80                   ({LINENUMBERSsimple.hs:4:1-4 }
81                    {Name: Main.main})
82                   ("Prefix")
83                   ("NoSrcStrict"))
84                  []
85                  ("GRHSs"
86                   ("NoExtField")
87                   [({LINENUMBERSsimple.hs:(4,6)-(7,41) }
88                     ("GRHS"
89                      ("NoExtField")
90                      []
91                      ({LINENUMBERSsimple.hs:(4,8)-(7,41) }
92                       ("HsDo"
93                        ("NoExtField")
94                        ("DoExpr")
95                        ({LINENUMBERSsimple.hs:(5,5)-(7,41) }
96                         [({LINENUMBERSsimple.hs:5:5-46 }
97                           ("BindStmt"
98                            ("NoExtField")
99                            ({LINENUMBERSsimple.hs:5:5-7 }
100                            ("VarPat"
101                             ("NoExtField")
102                             ({LINENUMBERSsimple.hs:5:5-7 }
103                              {Name: foo_ajn})))
104                           ({LINENUMBERSsimple.hs:5:12-46 }
105                            ("HsApp"
106                             ("NoExtField")
107                             ({LINENUMBERSsimple.hs:5:12-19 }
108                              ("HsVar"
109                               ("NoExtField")
110                               ({LINENUMBERSsimple.hs:5:12-19 }
111                                {Name: System.IO.putStrLn})))
112                             ({LINENUMBERSsimple.hs:5:21-46 }
113                              ("HsLit"
114                               ("NoExtField")
115                               ("HsString"
116                                ("SourceText"
117                                 "\"Hello, what's your name?\"")
```

```
118                                 {"FastString:" "FASTSTRING"}))))))
119                             ("SyntaxExpr"
120                              ("HsVar"
121                               ("NoExtField")
122                               ({LINENUMBERS<no location info> }
123                                {Name: GHC.Base.>>=}))
124                              []
125                              ("WpHole"))
126                             ("SyntaxExpr"
127                              ("HsLit"
128                               ("NoExtField")
129                               ("HsString"
130                                ("NoSourceText")
131                                {"FastString:" "FASTSTRING"}))
132                              []
133                              ("WpHole"))))
134                           ,({LINENUMBERSsimple.hs:6:5-19 }
135                             ("BindStmt"
136                              ("NoExtField")
137                              ({LINENUMBERSsimple.hs:6:5-8 }
138                               ("VarPat"
139                                ("NoExtField")
140                                ({LINENUMBERSsimple.hs:6:5-8 }
141                                 {Name: name_ajo})))
142                              ({LINENUMBERSsimple.hs:6:13-19 }
143                               ("HsVar"
144                                ("NoExtField")
145                                ({LINENUMBERSsimple.hs:6:13-19 }
146                                 {Name: System.IO.getLine})))
147                              ("SyntaxExpr"
148                               ("HsVar"
149                                ("NoExtField")
150                                ({LINENUMBERS<no location info> }
151                                 {Name: GHC.Base.>>=}))
152                               []
153                               ("WpHole"))
154                              ("SyntaxExpr"
155                               ("HsLit"
156                                ("NoExtField")
```

```
157                              ("HsString"
158                               ("NoSourceText")
159                               {"FastString:" "FASTSTRING"}))
160                              []
161                              ("WpHole"))))
162                            ,({LINENUMBERSsimple.hs:7:5-41 }
163                              ("LastStmt"
164                               ("NoExtField")
165                               ({LINENUMBERSsimple.hs:7:5-41 }
166                                ("HsApp"
167                                 ("NoExtField")
168                                 ({LINENUMBERSsimple.hs:7:5-12 }
169                                  ("HsVar"
170                                   ("NoExtField")
171                                   ({LINENUMBERSsimple.hs:7:5-12 }
172                                    {Name: System.IO.putStrLn})))
173                                 ({LINENUMBERSsimple.hs:7:14-41 }
174                                  ("HsPar"
175                                   ("NoExtField")
176                                   ({LINENUMBERSsimple.hs:7:15-40 }
177                                    ("OpApp"
178                                     {Fixity: infixr 5}
179                                     ({LINENUMBERSsimple.hs:7:15-20 }
180                                      ("HsLit"
181                                       ("NoExtField")
182                                       ("HsString"
183                                        ("SourceText"
184                                         "\"Hi, \"")
185                                        {"FastString:" "FASTSTRING"})))
186                                     ({LINENUMBERSsimple.hs:7:22-23 }
187                                      ("HsVar"
188                                       ("NoExtField")
189                                       ({LINENUMBERSsimple.hs:7:22-23 }
190                                        {Name: GHC.Base.++})))
191                                     ({LINENUMBERSsimple.hs:7:25-40 }
192                                      ("HsApp"
193                                       ("NoExtField")
194                                       ({LINENUMBERSsimple.hs:7:25-34 }
195                                        ("HsVar"
```

```
                                        ("NoExtField")
                                        ({LINENUMBERSsimple.hs:7:25-34 }
                                         {Name: Main.addexclaim})))
                                      ({LINENUMBERSsimple.hs:7:35-40 }
                                       ("HsPar"
                                        ("NoExtField")
                                        ({LINENUMBERSsimple.hs:7:36-39 }
                                         ("HsVar"
                                          ("NoExtField")
                                          ({LINENUMBERSsimple.hs:7:36-39 }
                                           {Name: name_ajo}))))))))))))
                             ("False")
                             ("SyntaxExpr"
                              ("HsLit"
                               ("NoExtField")
                               ("HsString"
                                ("NoSourceText")
                                {"FastString:" "FASTSTRING"}))
                              []
                              ("WpHole"))))])))))]
                   ({LINENUMBERS<no location info> }
                    ("EmptyLocalBinds"
                     ("NoExtField"))))))])
              ("FromSource"))
             ("WpHole")
             []))])))]
      [({LINENUMBERSsimple.hs:1:1-30 }
        ("TypeSig"
         ("NoExtField")
         [({LINENUMBERSsimple.hs:1:1-10 }
           {Name: Main.addexclaim})]
         ("HsWC"
          []
          ("HsIB"
           []
           ({LINENUMBERSsimple.hs:1:15-30 }
            ("HsFunTy"
             ("NoExtField")
             ({LINENUMBERSsimple.hs:1:15-20 }
```

```
235              ("HsTyVar"
236               ("NoExtField")
237               ("NotPromoted")
238               ({LINENUMBERSsimple.hs:1:15-20 }
239                {Name: GHC.Base.String})))
240             ({LINENUMBERSsimple.hs:1:25-30 }
241              ("HsTyVar"
242               ("NoExtField")
243               ("NotPromoted")
244               ({LINENUMBERSsimple.hs:1:25-30 }
245                {Name: GHC.Base.String}))))))))))]))
246    []
247    []
248    []
249    []
250    []
251    []
252    []
253    []
254    []
255    [])
256   [({LINENUMBERSsimple.hs:1:1 }
257     ("ImportDecl"
258      ("NoExtField")
259      ("NoSourceText")
260      ({LINENUMBERSsimple.hs:1:1 }
261       {ModuleName: Prelude})
262      ("Nothing")
263      ("False")
264      ("False")
265      ("NotQualified")
266      ("True")
267      ("Nothing")
268      ("Nothing")))]
269   ("Just"
270    [("(,)"
271      ({LINENUMBERS<no location info> }
272       ("IEVar"
273        ("NoExtField")
```

```
274        ({LINENUMBERS<no location info> }
275         ("IEName"
276          ({LINENUMBERS<no location info> }
277           {Name: Main.main})))))
278      [("Avail"
279        {Name: Main.main})])])
280   ("Nothing")))
```

Appendix 4: Syntax Features Collected for Graph Neural Networks of the Packages' Functions

(,)	DecBrL	HasDollar
-BACKSLASH-NUL-	DefD	HasParens
:%	DefaultDecl	Header
ActiveAfter	DeprecatedTxt	HsAnnotation
AlwaysActive	DerivD	HsAppTy
And	DerivDecl	HsAppType
AnnD	DoExpr	HsApp
AnyclassStrategy	EmptyLocalBinds	HsBangTy
ArithSeq	Exact	HsBoxedOrConstraintTuple
AsPat	ExpBr	HsBracket
AvailTC	ExplicitBidirectional	HsCase
Avail	ExplicitList	HsCharPrim
BangPat	ExplicitTuple	HsChar
BindStmt	ExprWithTySig	HsDataDefn
BodyStmt	FL	HsDerivingClause
Boxed	False	HsDo
CCallConv	FamDecl	HsExplicitListTy
CFunction	FamEqn	HsForAllTy
CImport	FamilyDecl	HsFractional
CLabel	FastString	HsFunTy
CONSTNUM	FieldLabel	HsGroup
CaseAlt	FieldOcc	HsIB
CatchAll	FixSig	HsIf
ClassDecl	FixitySig	HsIntegral
ClassOpSig	Fixity	HsInt{64
ClosedTypeFamily	ForD	HsIsString
ClsInstD	ForallInvis	HsLamCase
ClsInstDecl	ForeignImport	HsLam
CompleteMatchSig	FromSource	HsLet
ConDeclField	FromThenTo	HsListTy
ConDeclGADT	FromThen	HsLit
ConDeclH98	FromTo	HsModule
ConPatIn	From	HsMultiIf
DataDeclRn	FunBind	HsNumTy
DataDecl	FunLike	HsOpTy
DataFamInstD	FunRhs	HsOverLit
DataFamInstDecl	GRHS	HsParTy
DataFamily	GRHSs	HsPar
DataType	Generated	HsQTvs

Appendix 4: Syntax Features Collected for Graph Neural Networks of the Packages' Functions

HsQualTy	Inlinable	NoUserInline
HsRecField	InlinePragma	Nominal
HsRecFields	InlineSig	NonRecursive
HsRecFld	Inline	None
HsRuleRn	InstD	NotPromoted
HsRule	IsPromoted	NotQualified
HsRules	Just	Nothing
HsSCC	KindSig	OccName:
HsSpliceE	KindedTyVar	OpApp
HsSpliceTy	LINENUMBERS	OpenTypeFamily
HsSrcBang	LambdaExpr	Or
HsStarTy	LastStmt	OverLit
HsStrTy	LazyPat	Overlappable
HsStringPrim	LetStmt	Overlapping
HsString	ListComp	Overlaps
HsTupleTy	ListPat	PRIME-INT-PRIME
HsTyLit	LitPat	PRIME-OTHER-PRIME
HsTyVar	MG	PRIME-SPEC-PRIME
HsUnboundVar	Match	PRIME-WORD-PRIME
HsUnboxedTuple	MinimalSig	PRIMEPRIME
HsUntypedSplice	Missing	PSB
HsValArg	ModuleAnnProvenance	ParPat
HsValBinds	ModuleName	Parens
HsVar	NPat	PatBind
HsWC	NValBinds	PatSynBind
HsWildCardTy	NameSet	PatSynSig
IEModuleContents	Name	PlayRisky
IEName	NegApp	PlaySafe
IEPattern	NeverActive	PrefixCon
IEThingAbs	NewType	Prefix
IEThingAll	NewtypeStrategy	Present
IEThingWith	NoExtField	QUOTES
IEVar	NoIEWildcard	Qual
IL	NoInline	QualifiedPre
ImplicitBidirectional	NoParens	RecCon
ImplicitSplice	NoSig	RecordCon
ImportDecl	NoSourceText	RecordUpd
InfixCon	NoSrcStrict	Recursive
Infix	NoSrcUnpack	RoleAnnotD

Appendix 4: Syntax Features Collected for Graph Neural Networks of the Packages' Functions

RoleAnnotDecl	TuplePat	WpHole
RuleBndrSig	TyCID	XValBindsLR
RuleBndr	TyClGroup	digit
RuleD	TyFamInstD	funbind
STSTRING	TyFamInstDecl	gnode
SectionL	TypBr	list
SectionR	TypeSig	root
SigD	Unambiguous	typesig
SigPat	Unboxed	{-# INLINE
SourceText	Unqual	{-# NOINLINE
SpecInstSig	UserTyVar	{Name: ()}
SpecSig	ValBinds	{Name: (,)}
SpliceD	ValID	{Name: :}
SpliceDecl	ValueAnnProvenance	{Name: GHC.Types. }
SrcStrict	VarBr	{Name: [ ]}
SrcUnpack	VarPat	{abstract:UnitId}
StaticTarget	Var	"FastString:" "FASTSTRING"
StockStrategy	ViewPat	
StringLiteral	WarningD	
SynDecl	WarningTxt	Bag(Located (HsBind GhcPs))
SyntaxExpr	Warning	
TrueExprHole	Warnings	Bag(Located (HsBind Name))
True	WildPat	

Bibliography

[1] [n. d.]. Effects of Genetic Algorithm Options. https://www.mathworks.com/help/gads/options-in-genetic-algorithm.html

[2] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M.F.P. O'Boyle, J. Thomson, M. Toussaint, and C.K.I. Williams. 2006. Using machine learning to focus iterative optimization. In *International Symposium on Code Generation and Optimization (CGO'06)*. 11 pp.–305. https://doi.org/10.1109/CGO.2006.37

[3] R. W. Allard, K. A. Wolf, and R. A. Zemlin. 1964. Some effects of the 6600 computer on language structures. *Commun. ACM* 7, 2 (feb 1964), 112–119. https://doi.org/10.1145/363921.363940

[4] F. E. Allen and J. Cocke. 1976. A program data flow analysis procedure. *Commun. ACM* 19, 3 (mar 1976), 137. https://doi.org/10.1145/360018.360025

[5] Bowen Alpern, Anonthy Cocchi, and David Grove. 2012. Some new approaches to partial inlining. In *Proceedings of the Sixth ACM Workshop on Virtual Machines and Intermediate Languages* (Tucson, Arizona, USA) *(VMIL '12)*. Association for Computing Machinery, New York, NY, USA, 39–48. https://doi.org/10.1145/2414740.2414749

[6] J.P. Anderson. 1964. A Note on Some Compiling Algorithms. In *Communications of the ACM*.

[7] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. OpenTuner: An Extensible Framework for Program Autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation* (Edmonton, AB, Canada) *(PACT '14)*. Association for Computing Machinery, New York, NY, USA, 303–316. https://doi.org/10.1145/2628071.2628092

[8] Jason Ansel, Yee Lok Wong, Cy Chan, Marek Olszewski, Alan Edelman, and Saman Amarasinghe. 2011. Language and compiler support for auto-tuning variable-accuracy algorithms. In *International Symposium on Code Generation and Optimization (CGO 2011)*. 85–96. https://doi.org/10.1109/CGO.2011.5764677

[9] A.W. Appel. 1992. Compiling with Continuations.

[10] Andrew W. Appel. 1994. Loop Headers in λ-Calculus or CPS. *Lisp Symb. Comput.* 7, 4 (dec 1994), 337–343. https://doi.org/10.1007/BF01018615

[11] Matthew Arnold, Stephen Fink, Vivek Sarkar, and Peter F. Sweeney. 2000. A comparative study of static and profile-based heuristics for inlining. In *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (DYNAMO '00)*. Association for Computing Machinery, New York, NY, USA, 52–64. https://doi.org/10.1145/351397.351416

[12] Matthew Arnold, Adam Welc, and V. T. Rajan. 2005. Improving virtual machine performance using a cross-run profile repository. *SIGPLAN Not.* 40, 10 (oct 2005), 297–311. https://doi.org/10.1145/1103845.1094835

[13] Amir H. Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. 2018. A Survey on Compiler Autotuning using Machine Learning. *Comput. Surveys* 51 (09 2018), 96:1–. https://doi.org/10.1145/3197978

[14] Amir Hossein Ashouri, Giovanni Mariani, Gianluca Palermo, Eunjung Park, John Cavazos, and Cristina Silvano. 2016. COBAYN: Compiler Autotuning Framework Using Bayesian Networks. *ACM Trans. Archit. Code Optim.* 13, 2, Article 21 (jun 2016), 25 pages. https://doi.org/10.1145/2928270

[15] Clement Baker-Finch, Kevin Glynn, and Simon Peyton Jones. 2004. Constructed product result analysis for Haskell. *J. Funct. Program.* 14 (03 2004), 211–245. https://doi.org/10.1017/S0956796803004751

[16] Shajulin Benedict, Rejitha R.S., Philipp Gschwandtner, Radu Prodan, and Thomas Fahringer. 2015. Energy Prediction of OpenMP Applications Using Random Forest Modeling Approach. https://doi.org/10.1109/IPDPSW.2015.12

[17] D. Blum, S.K. Brown, A.G. Calavano, H.O. Hempy, and J. Suez. 1971. Current Technologies in FORTRAN Object Code Optimizations. In *International Business Machines Corporation, TR 00.2240*.

[18] François Bodin, Toru Kisuki, Peter Knijnenburg, Mike O' Boyle, and Erven Rohou. 1998. Iterative compilation in a non-linear optimisation space. In *Workshop on Profile and Feedback-Directed Compilation*. Paris, France. https://inria.hal.science/inria-00475919

[19] William J. Bowman, Swaha Miller, Vincent St-Amour, and R. Kent Dybvig. 2015. Profile-guided meta-programming. *SIGPLAN Not.* 50, 6 (jun 2015), 403–412. https://doi.org/10.1145/2813885.2737990

[20] Michael M. Bronstein, Joan Bruna, Taco Cohen, and Petar Veličković. 2021. Geometric Deep Learning: Grids, Groups, Graphs, Geodesics, and Gauges. arXiv:2104.13478 [cs.LG] https://arxiv.org/abs/2104.13478

[21] M. A. Bulyonkov. 1984. Polyvariant mixed computation for analyzer programs. *Acta Informatica* 21 (1984), 3–47. Issue 5. https://doi.org/10.1007/BF00271642

[22] Deborah R. Carvalho and Alex A. Freitas. 2004. A hybrid decision tree/genetic algorithm method for data mining. *Information Sciences* 163, 1 (2004), 13–35. https://doi.org/10.1016/j.ins.2003.03.013 Soft Computing Data Mining.

[23] John Cavazos, Christophe Dubach, Felix Agakov, Edwin Bonilla, Michael F. P. O'Boyle, Grigori Fursin, and Olivier Temam. 2006. Automatic performance model construction for the fast software exploration of new hardware designs. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems* (Seoul, Korea) *(CASES '06)*. Association for Computing Machinery, New York, NY, USA, 24–34. https://doi.org/10.1145/1176760.1176765

[24] John Cavazos, Grigori Fursin, Felix Agakov, Edwin Bonilla, Michael F.P. O'Boyle, and Olivier Temam. 2007. Rapidly Selecting Good Compiler Optimizations using Performance Counters. In *International Symposium on Code Generation and Optimization (CGO'07)*. 185–197. https://doi.org/10.1109/CGO.2007.32

[25] John Cavazos and J. Eliot B. Moss. 2004. Inducing heuristics to decide whether to schedule. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation* (Washington DC, USA) *(PLDI '04)*. Association for Computing Machinery, New York, NY, USA, 183–194. https://doi.org/10.1145/996841.996864

[26] John Cavazos and Michael F. P. O'Boyle. 2005. Automatic Tuning of Inlining Heuristics. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing (SC '05)*. IEEE Computer Society, USA, 14. https://doi.org/10.1109/SC.2005.14

[27] John Cavazos and Michael F. P. O'Boyle. 2006. Method-specific dynamic compilation using logistic regression. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications* (Portland, Oregon, USA) *(OOPSLA '06)*. Association for Computing Machinery, New York, NY, USA, 229–240. https://doi.org/10.1145/1167473.1167492

[28] G. J. Chaitin. 1982. Register allocation & spilling via graph coloring. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction* (Boston, Massachusetts, USA) *(SIGPLAN '82)*. Association for Computing Machinery, New York, NY, USA, 98–105. https://doi.org/10.1145/800230.806984

[29] D.R. Chakrabarti and Shin-Ming Liu. 2006. Inline analysis: beyond selection heuristics. In *International Symposium on Code Generation and Optimization (CGO'06)*. 12 pp.–232. https://doi.org/10.1109/CGO.2006.17

[30] Pohua P. Chang, Scott A. Mahlke, William Y. Chen, and Wen-Mei W. Hwu. 1992. Profile-guided automatic inline expansion for C programs. *Software: Practice and Experience* 22, 5 (1992), 349–369. https://doi.org/10.1002/spe.4380220502 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380220502

[31] Xuan Chen and Shun Long. 2009. Adaptive Multi-versioning for OpenMP Parallelization via Machine Learning. In *2009 15th International Conference on Parallel and Distributed Systems*. 907–912. https://doi.org/10.1109/ICPADS.2009.77

[32] Yang Chen, Yuanjie Huang, Lieven Eeckhout, Grigori Fursin, Liang Peng, Olivier Temam, and Chengyong Wu. 2010. Evaluating Iterative Optimization across 1000 Datasets. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada) *(PLDI '10)*. Association for Computing Machinery, New York, NY, USA, 448–459. https://doi.org/10.1145/1806596.1806647

[33] Peng-Fei Chuang, Howard Chen, Gerolf Hoflehner, Daniel Lavery, and Wei-Chung Hsu. 2008. Dynamic Profile Driven Code Version Selection.

[34] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. *SIGPLAN Not.* 35, 9 (Sept. 2000), 268–279. https://doi.org/10.1145/357766.351266

[35] John Cocke. 1970. Global common subexpression elimination. In *Proceedings of a Symposium on Compiler Optimization* (Urbana-Champaign, Illinois). Association for Computing Machinery, New York, NY, USA, 20–24. https://doi.org/10.1145/800028.808480

[36] The Haskell Community. 2024. *Hackage: The Haskell Package Repository*. https: https://hackage.haskell.org/ Accessed: 2024-17-06.

[37] Keith Cooper, Timothy Harvey, and Todd Waterman. 2008. An Adaptive Strategy for Inline Substitution, Vol. 4959. 69–84. https://doi.org/10.1007/978-3-540-78791-4_5

[38] Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. 1999. Optimizing for Reduced Code Space Using Genetic Algorithms. In *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems* (Atlanta, Georgia, USA) *(LCTES '99)*. Association for Computing Machinery, New York, NY, USA, 1–9. https://doi.org/10.1145/314403.314414

[39] International Business Machines Corporation. 1954. The IBM Mathematical Formula Translating System, FORTRAN. (1954).

[40] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. 2017. End-to-End Deep Learning of Optimization Heuristics. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 219–232. https://doi.org/10.1109/PACT.2017.24

[41] Olivier Danvy and Ulrik P. Schultz. 1998. Lambda-Dropping: Transforming Recursive Equations into Programs with Block Structure. *BRICS Report Series* 5, 54 (Dec. 1998). https://doi.org/10.7146/brics.v5i54.21959

[42] Dibyendu Das, Shahid Asghar Ahmad, and Venkataramanan Kumar. 2020. Deep Learning-based Approximate Graph-Coloring Algorithm for Register Allocation. In *2020 IEEE/ACM 6th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC) and Workshop on Hierarchical Parallelism for Exascale Computing (HiPar)*. 23–32. https://doi.org/10.1109/LLVMHPCHiPar51896.2020.00008

[43] Jack W. Davidson and Christopher W. Fraser. 1984. Automatic generation of peephole optimizations. *SIGPLAN Not.* 19, 6 (jun 1984), 111–116. https://doi.org/10.1145/502949.502885

[44] Jack W. Davidson and Anne M. Holler. 1988. A study of a C function inliner. *Softw. Pract. Exper.* 18, 8 (aug 1988), 775–790. https://doi.org/10.1002/spe.4380180805

[45] Jack W. Davidson and Anne M. Holler. 1988. A study of a C function inliner. *Software: Practice and Experience* 18, 8 (1988), 775–790. https://doi.org/10.1002/spe.4380180805 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380180805

[46] Jeffrey Dean and Craig Chambers. 1994. Towards better inlining decisions using inlining trials. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming* (Orlando, Florida, USA) *(LFP '94)*. Association for Computing Machinery, New York, NY, USA, 273–282. https://doi.org/10.1145/182409.182489

[47] Prasad Deshpande and Amit Somani. 1995. A Study and Analysis of Function Inlining. (12 1995).

[48] Mehrdad Dianati, In-Soo Song, and Mark Treiber. 2002. An Introduction to Genetic Algorithms and Evolution. https://api.semanticscholar.org/CorpusID:10975919

[49] K. Driesen and U. Holzle. 1998. The cascaded predictor: economical and adaptive branch target prediction. In *Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture*. 249–258. https://doi.org/10.1109/MICRO.1998.742786

[50] Christophe Dubach, John Cavazos, Björn Franke, Grigori Fursin, Michael F.P. O'Boyle, and Olivier Temam. 2007. Fast compiler optimisation evaluation using code-feature based performance prediction. In *Proceedings of the 4th International Conference on Computing Frontiers* (Ischia, Italy) *(CF '07)*. Association for Computing Machinery, New York, NY, USA, 131–142. https://doi.org/10.1145/1242531.1242553

[51] Richard Eisenberg. 2020. System FC, as implemented by GHC. (05 2020).

[52] J. Eliot, Eliot Moss, Paul Utgoff, John Cavazos, Doina Precup, Darko C, Carla Brodley, and David Scheeff. 1997. Learning to Schedule Straight-Line Code. (10 1997).

[53] Wallace J. Fleming, P. 1986. How not to lie with statistics: the correct way to summarize benchmark results.

[54] Robert W. Floyd. 1961. An algorithm for coding efficient arithmetic operations. *Commun. ACM* 4 (1961), 42–51. https://api.semanticscholar.org/CorpusID:7288583

[55] The Haskell Foundation. 2024. *Stackage: Stable Haskell package sets*. https://www.stackage.org/ Accessed: 2024-17-06.

[56] Inc. Free Software Foundation. 2024. *GCC, the GNU Compiler Collection*. https://gcc.gnu.org/onlinedocs/gcc-13.2.0/gcc/Instrumentation-Options.html

[57] Archana Ganapathi, Kaushik Datta, Armando Fox, and David Patterson. 2009. A case for machine learning to optimize multicore performance. In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism* (Berkeley, California) *(HotPar'09)*. USENIX Association, USA, 1.

[58] Stefan Ganser, Armin Grösslinger, Norbert Siegmund, Sven Apel, and Christian Lengauer. 2017. Iterative Schedule Optimization for Parallelization in the Polyhedron Model. *ACM Trans. Archit. Code Optim.* 14, 3, Article 23 (Aug. 2017), 26 pages. https://doi.org/10.1145/3109482

[59] Unai Garciarena and Roberto Santana. 2016. Evolutionary Optimization of Compiler Flag Selection by Learning and Exploiting Flags Interactions. In *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion* (Denver, Colorado, USA) *(GECCO '16 Companion)*. Association for Computing Machinery, New York, NY, USA, 1159–1166. https://doi.org/10.1145/2908961.2931696

[60] Eduardo C. Garrido-Merchán and Daniel Hernández-Lobato. 2020. Dealing with categorical and integer-valued variables in Bayesian Optimization with Gaussian processes. *Neurocomputing* 380 (2020), 20–35. https://doi.org/10.1016/j.neucom.2019.11.004

[61] Giorgis Georgakoudis, Konstantinos Parasyris, Chunhua Liao, David Beckingsale, Todd Gamblin, and Bronis de Supinski. 2023. Machine Learning-Driven Adaptive OpenMP For Portable Performance on Heterogeneous Systems. arXiv:2303.08873 [cs.PL] https://arxiv.org/abs/2303.08873

[62] Perry Gibson and José Cano. 2023. Transfer-Tuning: Reusing Auto-Schedules for Efficient Tensor Program Code Generation. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques* (Chicago, Illinois) *(PACT '22)*. Association for Computing Machinery, New York, NY, USA, 28–39. https://doi.org/10.1145/3559009.3569682

[63] Thomas Gilray, J. R. King, and Matthew Might. 2014. Partitioning 0-CFA for the GPU. https://api.semanticscholar.org/CorpusID:10462195

[64] Dominik Grewe, Zheng Wang, and Michael F. P. O'Boyle. 2013. Portable mapping of data parallel programs to OpenCL for heterogeneous systems. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 1–10. https://doi.org/10.1109/CGO.2013.6494993

[65] David Grove, Jeffrey Dean, Charles Garrett, and Craig Chambers. 1995. Profile-guided receiver class prediction. *SIGPLAN Not.* 30, 10 (oct 1995), 108–123. https://doi.org/10.1145/217839.217848

[66] Priya Gupta, Aditya Jha, Brinda Gupta, Kime Sumpi, Sabyasachi Sahoo, and Mukkoti Maruthi Venkata Chalapathi. 2023. Techniques and Trade-Offs in Function Inlining Optimization. *EAI Endorsed Transactions on Scalable Information Systems* 11, 4 (Nov. 2023). https://doi.org/10.4108/eetsis.4453

[67] Ameer Haj-Ali, Nesreen K. Ahmed, Ted Willke, Yakun Sophia Shao, Krste Asanovic, and Ion Stoica. 2020. NeuroVectorizer: end-to-end vectorization with deep reinforcement learning. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization* (San Diego, CA, USA) *(CGO 2020)*. Association for Computing Machinery, New York, NY, USA, 242–255. https://doi.org/10.1145/3368826.3377928

[68] Ameer Haj-Ali, Qijing (Jenny) Huang, John Xiang, William Moses, Krste Asanovic, John Wawrzynek, and Ion Stoica. 2020. AutoPhase: Juggling HLS Phase Orderings in Random Forests with Deep Reinforcement Learning. In *Proceedings of Machine Learning and Systems*, I. Dhillon, D. Papailiopoulos, and V. Sze (Eds.), Vol. 2. 70–81. https://proceedings.mlsys.org/paper_files/paper/2020/file/5b47430e24a5a1f9fe21f0e8eb814131-Paper.pdf

[69] William L. Hamilton. 2020. *The Graph Neural Network Model*. Springer International Publishing, Cham, 51–70. https://doi.org/10.1007/978-3-031-01588-5_5

[70] haskell.org. 2024. Haskell.org:Modules. https://www.haskell.org/onlinereport/haskell2010/haskellch5.html

[71] HaskellWiki. 2017. Inlining and Specialisation — HaskellWiki,. https://wiki.haskell.org/index.php?title=Inlining_and_Specialisation&oldid=61640 [Online; accessed 8-June-2024].

[72] K. Hazelwood and D. Grove. 2003. Adaptive online context-sensitive inlining. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.* 253–264. https://doi.org/10.1109/CGO.2003.1191550

[73] Jeff Heaton. 2017. The Number of Hidden Layers. https://www.heatonresearch.com/2017/06/01/hidden-layers.html

[74] Nevin Heintze and David McAllester. 1997. Linear-time subtransitive control flow analysis. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation* (Las Vegas, Nevada, USA) *(PLDI '97)*. Association for Computing Machinery, New York, NY, USA, 261–272. https://doi.org/10.1145/258915.258939

[75] Erik Hellsten, Artur Souza, Johannes Lenfers, Rubens Lacouture, Olivia Hsu, Adel Ejjeh, Fredrik Kjolstad, Michel Steuwer, Kunle Olukotun, and Luigi Nardi. 2022. BaCO: A Fast and Portable Bayesian Compiler Optimization Framework. https://doi.org/10.48550/arXiv.2212.11142

[76] Steven C.H. Hoi, Doyen Sahoo, Jing Lu, and Peilin Zhao. 2021. Online learning: A comprehensive survey. *Neurocomputing* 459 (2021), 249–289. https://doi.org/10.1016/j.neucom.2021.04.112

[77] Celeste Hollenbeck, Michael F. P. O'Boyle, and Michel Steuwer. 2022. Investigating magic numbers: improving the inlining heuristic in the Glasgow Haskell Compiler. In *Proceedings of the 15th ACM SIGPLAN International Haskell Symposium* (Ljubljana, Slovenia) *(Haskell 2022)*. Association for Computing Machinery, New York, NY, USA, 81–94. https://doi.org/10.1145/3546189.3549918

[78] Celeste Hollenbeck and Michael F. P. O'Boyle. 2024. Hot Call-Chain Inlining for the Glasgow Haskell Compiler. In *Proceedings of the 23rd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (Pasadena, CA, USA) *(GPCE '24)*. Association for Computing Machinery, New York, NY, USA, 66–79. https://doi.org/10.1145/3689484.3690730

[79] Kenneth Hoste and Lieven Eeckhout. 2008. Cole: compiler optimization level exploration. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Boston, MA, USA) *(CGO '08)*. Association for Computing Machinery, New York, NY, USA, 165–174. https://doi.org/10.1145/1356058.1356080

[80] Kenneth Hoste, Aashish Phansalkar, Lieven Eeckhout, Andy Georges, Lizy K. John, and Koen De Bosschere. 2006. Performance prediction based on inherent program similarity. In *2006 International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 114–122.

[81] Suresh Jagannathan and Andrew Wright. 1996. Flow-directed inlining. *SIGPLAN Not.* 31, 5 (may 1996), 193–205. https://doi.org/10.1145/249069.231417

[82] Shalini Jain, s Venkatakeerthy, Rohit Aggarwal, Tharun Dangeti, Dibyendu Das, and Ramakrishna Upadrasta. 2022. Reinforcement Learning assisted Loop Distribution for Locality and Vectorization. https://doi.org/10.1109/LLVM-HPC56686.2022.00006

[83] Michael R. Jantz and Prasad A. Kulkarni. 2013. Exploring single and multilevel JIT compilation policy for modern machines 1. *ACM Trans. Archit. Code Optim.* 10, 4, Article 22 (dec 2013), 29 pages. https://doi.org/10.1145/2541228.2541229

[84] Yunlian Jiang, Eddy Z. Zhang, Kai Tian, Feng Mao, Malcom Gethers, Xipeng Shen, and Yaoqing Gao. 2010. Exploiting statistical correlations for proactive prediction of program behaviors. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Toronto, Ontario, Canada) *(CGO '10)*. Association for Computing Machinery, New York, NY, USA, 248–256. https://doi.org/10.1145/1772954.1772989

[85] Thomas Johnsson. 1985. Lambda lifting: Transforming programs to recursive equations. In *Functional Programming Languages and Computer Architecture*, Jean-Pierre Jouannaud (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 190–203.

[86] Neil D. Jones and Steven S. Muchnick. 1979. Flow analysis and optimization of LISP-like structures. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (San Antonio, Texas) *(POPL '79)*. Association for Computing Machinery, New York, NY, USA, 244–256. https://doi.org/10.1145/567752.567776

[87] Owen Kaser and C.R. Ramakrishnan. 1998. Evaluating inlining techniques. *Computer Languages* 24, 2 (1998), 55–72. https://doi.org/10.1016/S0096-0551(98)00003-4

[88] Gary A. Kildall. 1973. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Boston, Massachusetts) *(POPL '73)*. Association for Computing Machinery, New York, NY, USA, 194–206. https://doi.org/10.1145/512927.512945

[89] Alexis King. [n. d.]. Making GHC Faster at Emitting Code. https://www.tweag.io/blog/2022-12-22-making-ghc-faster-at-emitting-code/ Accessed: 2024-04-30.

[90] T. Kisuki, P.M.W. Knijnenburg, and M.F.P. O'Boyle. 2000. Combined selection of tile sizes and unroll factors using iterative compilation. In *Proceedings 2000 International Conference on Parallel Architectures and Compilation Techniques (Cat. No.PR00622)*. 237–246. https://doi.org/10.1109/PACT.2000.888348

[91] Edward Kmett. 2019. *INLINE /= INLINABLE*. https://www.reddit.com/r/haskell/comments/cjkc3l/should_i_be_inlining_instance_implementations/ Accessed: 2024-03-06.

[92] Sameer Kulkarni and John Cavazos. 2012. Mitigating the compiler optimization phase-ordering problem using machine learning. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Tucson, Arizona, USA) *(OOPSLA '12)*. Association for Computing Machinery, New York, NY, USA, 147–162. https://doi.org/10.1145/2384616.2384628

[93] Sameer Kulkarni, John Cavazos, Christian Wimmer, and Douglas Simon. 2013. Automatic construction of inlining heuristics using machine learning. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 1–12. https://doi.org/10.1109/CGO.2013.6495004

[94] Georgios Lappas. 2007. Estimating the Size of Neural Networks from the Number of Available Training Data. In *Artificial Neural Networks – ICANN 2007*, Joaquim Marques de Sá, Luís A. Alexandre, Włodzisław Duch, and Danilo Mandic (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 68–77.

[95] Samuel Larsen and Saman Amarasinghe. 2000. Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (Vancouver, British Columbia, Canada) *(PLDI '00)*. Association for Computing Machinery, New York, NY, USA, 145–156. https://doi.org/10.1145/349299.349320

[96] Jeremy Lau, Matthew Arnold, Michael Hind, and Brad Calder. 2006. Online performance auditing: using hot optimizations without getting burned. *SIGPLAN Not.* 41, 6 (June 2006), 239–251. https://doi.org/10.1145/1133255.1134010

[97] Hugh Leather, Edwin Bonilla, and Michael O'boyle. 2014. Automatic feature generation for machine learning–based optimising compilation. *ACM Trans. Archit. Code Optim.* 11, 1, Article 14 (Feb. 2014), 32 pages. https://doi.org/10.1145/2536688

[98] R. Leupers and P. Marwedel. 1999. Function inlining under code size constraints for embedded processors. In *1999 IEEE/ACM International Conference on Computer-Aided Design. Digest of Technical Papers (Cat. No.99CH37051)*. 253–256. https://doi.org/10.1109/ICCAD.1999.810657

[99] Yang Liu, Wissam M. Sid-Lakhdar, Osni Marques, Xinran Zhu, Chang Meng, James W. Demmel, and Xiaoye S. Li. 2021. GPTune: multitask learning for autotuning exascale applications. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Virtual Event, Republic of Korea) *(PPoPP '21)*. Association for Computing Machinery, New York, NY, USA, 234–246. https://doi.org/10.1145/3437801.3441621

[100] Yixun Liu, Eddy Z. Zhang, and Xipeng Shen. 2009. A cross-input adaptive framework for GPU program optimizations. In *2009 IEEE International Symposium on Parallel Distributed Processing*. 1–10. https://doi.org/10.1109/IPDPS.2009.5160988

[101] Paul Lokuciejewski, Fatih Gedikli, Peter Marwedel, and Katharina Morik. 2012. K.: Automatic WCET Reduction by Machine Learning Based Heuristics for Function Inlining. (05 2012).

[102] Yulong Luo, Guangming Tan, Zeyao Mo, and Ninghui Sun. 2015. FAST: A Fast Stencil Autotuning Framework Based On An Optimal-solution Space Model. In *Proceedings of the 29th ACM on International Conference on Supercomputing* (Newport Beach, California, USA) *(ICS '15)*. Association for Computing Machinery, New York, NY, USA, 187–196. https://doi.org/10.1145/2751205.2751214

[103] Alberto Magni, Christophe Dubach, and Michael O'Boyle. 2014. Automatic optimization of thread-coarsening for graphics processors. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation* (Edmonton, AB, Canada) *(PACT '14)*. Association for Computing Machinery, New York, NY, USA, 455–466. https://doi.org/10.1145/2628071.2628087

[104] Zoltan Majo, Tobias Hartmann, Marcel Mohler, and Thomas R. Gross. 2017. Integrating Profile Caching into the HotSpot Multi-Tier Compilation System. In *Proceedings of the 14th International Conference on Managed Languages and Runtimes* (Prague, Czech Republic) *(ManLang 2017)*. Association for Computing Machinery, New York, NY, USA, 105–118. https://doi.org/10.1145/3132190.3132210

[105] Simon Marlow. 2005. *Haskell Cafe post by Simon Marlow*. https://haskell-cafe.haskell.narkive.com/jJTv7WgN/proposal-habench-a-haskell-benchmark-suite#post3 Accessed: 2021-04-06.

[106] W. M. McKeeman. 1965. Peephole optimization. *Commun. ACM* 8, 7 (jul 1965), 443–444. https://doi.org/10.1145/364995.365000

[107] Charith Mendis, Cambridge Yang, Yewen Pu, Saman Amarasinghe, and Michael Carbin. 2019. *Compiler auto-vectorization with imitation learning*. Curran Associates Inc., Red Hook, NY, USA.

[108] Jan Midtgaard. 2012. Control-flow analysis of functional programs. *ACM Comput. Surv.* 44, 3, Article 10 (jun 2012), 33 pages. https://doi.org/10.1145/2187671.2187672

[109] Frederic P. Miller, Agnes F. Vandome, and John McBrewster. 2009. *Magic Number (Programming): Computer Programming, File Format, Software Documentation, Globally Unique Identifier, Enumerated Type, Hexspeak, NaN, Version ... OSCAR Protocol, AOL Instant Messenger*. Alpha Press.

[110] R.C. Miller and B.J. Oldfield. 1956. Producing Computer Instructions for the PACT I Compiler. (1956).

[111] Yaron Minsky. 2016. *A better inliner for OCaml, and why it matters*. https://blog.janestreet.com/flambda/

[112] Antoine Monsifrot, François Bodin, and René Quiniou. 2002. A Machine Learning Approach to Automatic Production of Compiler Heuristics. In *Artificial Intelligence: Methodology, Systems, and Applications*, Donia Scott (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 41–50.

[113] Raphael Mosaner, David Leopoldseder, Wolfgang Kisling, Lukas Stadler, and Hanspeter Mössenböck. 2022. Compilation Forking: A Fast and Flexible Way of Generating Data for Compiler-Internal Machine Learning Tasks. *Art Sci. Eng. Program.* 7 (2022), 3. https://api.semanticscholar.org/CorpusID:250089423

[114] Raphael Mosaner, David Leopoldseder, Wolfgang Kisling, Lukas Stadler, and Hanspeter Mössenböck. 2022. Machine-Learning-Based Self-Optimizing Compiler Heuristics. In *Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes* (Brussels, Belgium) *(MPLR '22)*. Association for Computing Machinery, New York, NY, USA, 98–111. https://doi.org/10.1145/3546918.3546921

[115] Raphael Mosaner, David Leopoldseder, Lukas Stadler, and Hanspeter Mössenböck. 2021. Using Machine Learning to Predict the Code Size Impact of Duplication Heuristics in a Dynamic Compiler. In *Proceedings of the 18th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes* (Münster, Germany) *(MPLR 2021)*. Association for Computing Machinery, New York, NY, USA, 127–135. https://doi.org/10.1145/3475738.3480943

[116] Jacob Murel and Eda Kavlakoglu. 2024. What is reinforcement learning? https://https://www.ibm.com/topics/reinforcement-learning

[117] Muhammad Nadeem, Polychronis Xekalakis, John Cavazos, and Marcelo Cintra. 2007. Using PredictiveModeling for Cross-Program Design Space Exploration in Multicore Systems. 327–338. https://doi.org/10.1109/PACT.2007.4336223

[118] Ikuo Nakata. 1967. On compiling algorithms for arithmetic expressions. *Commun. ACM* 10, 8 (aug 1967), 492–494. https://doi.org/10.1145/363534.363549

[119] Mircea Namolaru, Albert Cohen, Grigori Fursin, Ayal Zaks, and Ari Freund. 2010. Practical aggregation of semantical program properties for machine learning based optimization. In *Proceedings of the 2010 International Conference on Compilers, Architectures and Synthesis for Embedded Systems* (Scottsdale, Arizona, USA) *(CASES '10)*. Association for Computing Machinery, New York, NY, USA, 197–206. https://doi.org/10.1145/1878921.1878951

[120] Erick Ochoa, Cijie Xia, Karim Ali, Andrew Craik, and José Nelson Amaral. 2021. *U Can't Inline This!* IBM Corp., USA, 173–182.

[121] Iaroslav Omelianenko. 2019. *Hands-On Neuroevolution with Python*. Packt Publishing.

[122] Eunjung Park, John Cavazos, and Marco A. Alvarez. 2012. Using graph-based program characterization for predictive modeling. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization* (San Jose, California) *(CGO '12)*. Association for Computing Machinery, New York, NY, USA, 196–206. https://doi.org/10.1145/2259016.2259042

[123] Will Partain. 1993. The nofib Benchmark Suite of Haskell Programs. In *Functional Programming, Glasgow 1992*, John Launchbury and Patrick Sansom (Eds.). Springer London, London, 195–202.

[124] WD Partain, A Santos, and Simon Peyton Jones. 1996. Let-floating: moving bindings to give faster programs. https://www.microsoft.com/en-us/research/publication/let-floating-moving-bindings-to-give-faster-programs/ ACM SIGPLAN International Conference on Functional Programming (ICFP'96).

[125] Alain Petrowski and Sana Ben-Hamida. 2017. *A Generic Evolutionary Algorithm*. Wiley, Hoboken, NJ.

[126] Simon Peyton Jones and Simon Marlow. 2002. Secrets of the Glasgow Haskell Compiler inliner. *Journal of Functional Programming* 12 (July 2002), 393–434. https://www.microsoft.com/en-us/research/publication/secrets-of-the-glasgow-haskell-compiler-inliner/

[127] Simon L. Peyton Jones and AndréL.M. Santos. 1998. A transformation-based optimiser for Haskell. *Science of Computer Programming* 32, 1 (1998), 3–47. https://doi.org/10.1016/S0167-6423(97)00029-4 6th European Symposium on Programming.

[128] M.T. Pickering. 2021. *Understanding the Interaction Between Elaboration and Quotation*. University of Bristol. https://books.google.co.uk/books?id=WzrGzgEACAAJ

[129] C.R. Pirnat, J.C.C. Han, K. Maruyama, R.M. Lefler, T. Nakagawa, and H.C. Lai. 1971. *Tree Height Reduction for Parallel Processing of Blocks of Fortran Assignment Statements*. Number nos. 488-494 in A Problem in Form Perception: Odd Shape Detection. Department of Computer Science, University of Illinois at Urbana-Champaign. https://books.google.co.uk/books?id=iao_xQEACAAJ

[130] Aleksandar Prokopec, Gilles Duboscq, David Leopoldseder, and Thomas Wïrthinger. 2019. An Optimization-Driven Incremental Inline Substitution Algorithm for Just-in-Time Compilers. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 164–179. https://doi.org/10.1109/CGO.2019.8661171

[131] C.V. Ramamoorthy and M.J. Gonzalez. 1971. Subexpression Ordering in the Execution of Arithmetic Expressions. In *Communications of the ACM*.

[132] Miha Ravber, Shih-Hsi Liu, Marjan Mernik, and Matej Črepinšek. 2022. Maximum number of generations as a stopping criterion considered harmful. *Applied Soft Computing* 128 (2022), 109478. https://doi.org/10.1016/j.asoc.2022.109478

[133] Gregor Richards, Andreas Gal, Brendan Eich, and Jan Vitek. 2011. Automated Construction of JavaScript Benchmarks. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Portland, Oregon, USA) *(OOPSLA '11)*. Association for Computing Machinery, New York, NY, USA, 677–694. https://doi.org/10.1145/2048066.2048119

[134] Stéphane Ross, Geoffrey J. Gordon, and J. Andrew Bagnell. 2010. A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning. In *International Conference on Artificial Intelligence and Statistics*. https://api.semanticscholar.org/CorpusID:103456

[135] Jaehun Ryu, Eunhyeok Park, and Hyojin Sung. 2022. One-shot tuner for deep learning compilers. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction* (Seoul, South Korea) *(CC 2022)*. Association for Computing Machinery, New York, NY, USA, 89–103. https://doi.org/10.1145/3497776.3517774

[136] Issa Saba, Eishi Arima, Dai Liu, and Martin Schulz. 2022. Orchestrated Co-scheduling, Resource Partitioning, and Power Capping on CPU-GPU Heterogeneous Systems via Machine Learning. In *Architecture of Computing Systems*, Martin Schulz, Carsten Trinitis, Nikela Papadopoulou, and Thilo Pionteck (Eds.). Springer International Publishing, Cham, 51–67.

[137] Harpreet Singh Sachdev. 2020. Choosing number of Hidden Layers and number of hidden neurons in Neural Networks. https://www.linkedin.com/pulse/choosing-number-hidden-layers-neurons-neural-networks-sachdev/

[138] Biplab Kumar Saha, Tiffany A. Connors, Saami Rahman, and Apan Qasem. 2017. A Machine Learning Approach to Automatic Creation of Architecture-Sensitive Performance Heuristics. In *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. 18–25. https://doi.org/10.1109/HPCC-SmartCity-DSS.2017.3

[139] Andre Santos and Simon Peyton Jones. 1995. *Compilation by transformation for non-strict functional languages*. Ph. D. Dissertation. https://www.microsoft.com/en-us/research/publication/compilation-transformation-non-strict-functional-languages/

[140] Urvij Saroliya, Eishi Arima, Dai Liu, and Martin Schulz. 2023. Hierarchical Resource Partitioning on Modern GPUs: A Reinforcement Learning Approach. In *2023 IEEE International Conference on Cluster Computing (CLUSTER)*. 185–196. https://doi.org/10.1109/CLUSTER52292.2023.00023

[141] Robert W. Scheifler. 1977. An analysis of inline substitution for a structured programming language. *Commun. ACM* 20, 9 (sep 1977), 647–654. https://doi.org/10.1145/359810.359830

[142] Paul B. Schneck. 1973. A survey of compiler optimization techniques. In *ACM Annual Conference*. https://api.semanticscholar.org/CorpusID:8725221

[143] Manuel Serrano. 1997. Inline Expansion: When and How?. In *PLILP*.

[144] Andreas Sewe, Jannik Jochem, and Mira Mezini. 2011. Next in Line, Please! Exploiting the Indirect Benefits of Inlining by Accurately Predicting Further Inlining. In *Proceedings of the Compilation of the Co-Located Workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, & VMIL'11* (Portland, Oregon, USA) *(SPLASH '11 Workshops)*. Association for Computing Machinery, New York, NY, USA, 317–328. https://doi.org/10.1145/2095050.2095102

[145] Hafsah Shahzad, Ahmed Sanaullah, Sanjay Arora, Robert Munafo, Xiteng Yao, Ulrich Drepper, and Martin Herbordt. 2022. Reinforcement Learning Strategies for Compiler Optimization in High level Synthesis. In *2022 IEEE/ACM Eighth Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*. 13–22. https://doi.org/10.1109/LLVM-HPC56686.2022.00007

[146] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. 2002. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California) *(ASPLOS X)*. Association for Computing Machinery, New York, NY, USA, 45–57. https://doi.org/10.1145/605397.605403

[147] Bhargav Shivkumar, Jeffrey C. Murphy, and Lukasz Ziarek. 2021. Real-time MLton: A Standard ML runtime for real-time functional programs. *J. Funct. Program.* 31 (2021), e19. https://doi.org/10.1017/S0956796821000174

[148] Karan Singh, Matthew Curtis-Maury, Sally McKee, Filip Blagojevic, Dimitrios Nikolopoulos, Bronis Supinski, and Martin Schulz. 2010. Comparing Scalability Prediction Strategies on an SMP of CMPs, Vol. 6271. 143–155. https://doi.org/10.1007/978-3-642-15277-1_14

[149] Cesar Soares. 2023. *How Tiered Compilation works in OpenJDK*. https://devblogs.microsoft.com/java/how-tiered-compilation-works-in-openjdk/

[150] K.O. Stanley and R. Miikkulainen. 2002. Efficient evolution of neural network topologies. In *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No.02TH8600)*, Vol. 2. 1757–1762 vol.2. https://doi.org/10.1109/CEC.2002.1004508

[151] Kenneth O. Stanley and Risto Miikkulainen. 2002. Evolving Neural Networks through Augmenting Topologies. *Evolutionary Computation* 10, 2 (2002), 99–127. https://doi.org/10.1162/106365602320169811

[152] M. Stephenson and Saman Amarasinghe. 2005. Predicting Unroll Factors Using Supervised Classification, Vol. 2005. 123– 134. https://doi.org/10.1109/CGO.2005.29

[153] Mark Stephenson, Saman Amarasinghe, Martin Martin, and Una-May O'Reilly. 2003. Meta Optimization: Improving Compiler Heuristics with Machine Learning. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '03)*. Association for Computing Machinery, New York, NY, USA, 77–90. https://doi.org/10.1145/781131.781141

[154] Harold S. Stone. 1967. One-Pass compilation of arithmetic expressions for a parallel processor. *Commun. ACM* 10, 4 (apr 1967), 220–223. https://doi.org/10.1145/363242.363256

[155] Felipe Petroski Such, Vashisht Madhavan, Edoardo Conti, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. 2018. Deep Neuroevolution: Genetic Algorithms Are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning. arXiv:1712.06567 [cs.NE]

[156] Toshio Suganuma, Toshiaki Yasue, and Toshio Nakatani. 2002. An Empirical Study of Method In-lining for a Java Just-in-Time Compiler. 91–104.

[157] Ben Taylor, Vicent Sanz Marco, and Zheng Wang. 2017. Adaptive optimization for OpenCL programs on embedded heterogeneous systems. In *Proceedings of the 18th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems* (Barcelona, Spain) *(LCTES 2017)*. Association for Computing Machinery, New York, NY, USA, 11–20. https://doi.org/10.1145/3078633.3081040

[158] Cabal Team. 2024. Cabal: A framework for packaging Haskell software. Distribution.Package. https://hackage.haskell.org/package/Cabal-1.8.0.6/docs/Distribution-Package.html

[159] Cabal Team. 2024. Cabal reference. https://cabal.readthedocs.io/en/3.4/index.html

[160] Cabal Team. 2024. Cabal reference: 5.9 Packages. https://downloads.haskell.org/ghc/9.10-latest/docs/users_guide/packages.html

[161] GHC Team. 2024. *Glasgow Haskell Compiler User's Guide: 6.19.1. Rewrite rules*. https://downloads.haskell.org/ghc/latest/docs/users_guide/exts/rewrite_rules.html#pragma-RULES

[162] GHC Team. 2024. *Glasgow Haskell Compiler User's Guide: Profiling*. https://downloads.haskell.org/ghc/latest/docs/users_guide/profiling.html

[163] GHC Team. 2024. Glasgow Haskell Compiler Users Guide: Using Optimisation. https://downloads.haskell.org/ghc/9.10-latest/docs/users_guide/using-optimisation.html

[164] PyG Team. 2024. "torch_geometric.nn/conv.MessagePassing". "https://pytorch-geometric.readthedocs.io/en/latest/generated/torch_geometric.nn.conv.MessagePassing.html#torch_geometric.nn.conv.MessagePassing" [Online; accessed 12-September-2024].

[165] The GHC Team. [n. d.]. The Glorious Glasgow Haskell Compilation System User's Guide.

[166] Theodoros Theodoridis, Tobias Grosser, and Zhendong Su. 2022. Understanding and exploiting optimal function inlining. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 977–989. https://doi.org/10.1145/3503222.3507744

[167] J. F. Thorlin. 1967. Code generation for PIE (Parallel Instruction Execution) computers. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference* (Atlantic City, New Jersey) *(AFIPS '67 (Spring))*. Association for Computing Machinery, New York, NY, USA, 641–643. https://doi.org/10.1145/1465482.1465585

[168] Nicolas Tollenaere, Guillaume Iooss, Stéphane Pouget, Hugo Brunie, Christophe Guillon, Albert Cohen, P. Sadayappan, and Fabrice Rastello. 2023. Autotuning Convolutions Is Easier Than You Think. *ACM Trans. Archit. Code Optim.* 20, 2, Article 20 (mar 2023), 24 pages. https://doi.org/10.1145/3570641

[169] Stephen Toub. 2021. *Performance Improvements in .NET 6*. https://devblogs.microsoft.com/dotnet/performance-improvements-in-net-6/#jit

[170] Mircea Trofin, Yundi Qian, Eugene Brevdo, Zinan Lin, Krzysztof Choromanski, and David Li. 2021. MLGO: a Machine Learning Guided Compiler Optimizations Framework.

[171] David Van Horn and Harry G. Mairson. 2008. Deciding kCFA is complete for EXPTIME. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming* (Victoria, BC, Canada) *(ICFP '08)*. Association for Computing Machinery, New York, NY, USA, 275–282. https://doi.org/10.1145/1411204.1411243

[172] Martijn van Otterlo and Marco Wiering. 2012. *Reinforcement Learning and Markov Decision Processes*. Springer Berlin Heidelberg, Berlin, Heidelberg, 3–42. https://doi.org/10.1007/978-3-642-27645-3_1

[173] Kapil Vaswani, Matthew J. Thazhuthaveetil, Y. N. Srikant, and P. J. Joseph. 2007. Microarchitecture Sensitive Empirical Models for Compiler Optimizations. In *International Symposium on Code Generation and Optimization (CGO'07)*. 131–143. https://doi.org/10.1109/CGO.2007.25

[174] S. VenkataKeerthy, Siddharth Jain, Anilava Kundu, Rohit Aggarwal, Albert Cohen, and Ramakrishna Upadrasta. 2023. RL4ReAl: Reinforcement Learning for Register Allocation. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction* (, Montréal, QC, Canada,) *(CC 2023)*. Association for Computing Machinery, New York, NY, USA, 133–144. https://doi.org/10.1145/3578360.3580273

[175] Bill Vorhies. 2017. "Have You Heard About Unsupervised Decision Trees". https://www.datasciencecentral.com/have-you-heard-about-unsupervised-decision-trees/ [Online; accessed 7-October-2024].

[176] Zheng Wang and Michael F.P. O'Boyle. 2009. Mapping parallelism to multi-cores: a machine learning based approach. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Raleigh, NC, USA) *(PPoPP '09)*. Association for Computing Machinery, New York, NY, USA, 75–84. https://doi.org/10.1145/1504176.1504189

[177] Zheng Wang and Michael F.P. O'Boyle. 2010. Partitioning streaming parallelism for multi-cores: a machine learning based approach. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques* (Vienna, Austria) *(PACT '10)*. Association for Computing Machinery, New York, NY, USA, 307–318. https://doi.org/10.1145/1854273.1854313

[178] Zheng Wang, Georgios Tournavitis, Björn Franke, and Michael F. P. O'boyle. 2014. Integrating profile-driven parallelism detection and machine-learning-based mapping. *ACM Trans. Archit. Code Optim.* 11, 1, Article 2 (Feb. 2014), 26 pages. https://doi.org/10.1145/2579561

[179] Ben Wegbreit. 1975. Property extraction in well-founded property sets. *IEEE Transactions on Software Engineering* SE-1, 3 (1975), 270–285. https://doi.org/10.1109/TSE.1975.6312852

[180] Mark N. Wegman and F. Kenneth Zadeck. 1991. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.* 13, 2 (apr 1991), 181–210. https://doi.org/10.1145/103135.103136

[181] Yuan Wen, Zheng Wang, and Michael F. P. O'Boyle. 2014. Smart multi-task scheduling for OpenCL programs on CPU/GPU heterogeneous platforms. In *2014 21st International Conference on High Performance Computing (HiPC)*. 1–10. https://doi.org/10.1109/HiPC.2014.7116910

[182] Tomofumi Yuki, Lakshminarayanan Renganarayanan, Sanjay Rajopadhye, Charles Anderson, Alexandre E. Eichenberger, and Kevin O'Brien. 2010. Automatic creation of tile size selection models. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Toronto, Ontario, Canada) *(CGO '10)*. Association for Computing Machinery, New York, NY, USA, 190–199. https://doi.org/10.1145/1772954.1772982

[183] Vladislav Zavialov. 2020. 10 Reasons to Use Haskell. https://serokell.io/blog/10-reasons-to-use-haskell

[184] Minjia Zhang, Menghao Li, Chi Wang, and Mingqin Li. 2021. DynaTune: Dynamic Tensor Program Optimization in Deep Neural Network Compilation. In *International Conference on Learning Representations*. https://openreview.net/forum?id=GTGb3M_KcUl

[185] Peng Zhang, Jianbin Fang, Tao Tang, Canqun Yang, and Zheng Wang. 2018. Auto-tuning Streamed Applications on Intel Xeon Phi. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 515–525. https://doi.org/10.1109/IPDPS.2018.00061

[186] Peng Zhao and José Amaral. 2003. To Inline or Not to Inline? Enhanced Inlining Decisions, Vol. 2958. 405–419. https://doi.org/10.1007/978-3-540-24644-2_26

[187] Shengtong Zhong, Yang Shen, and Fei Hao. 2009. Tuning Compiler Optimization Options via Simulated Annealing. In *2009 Second International Conference on Future Information Technology and Management Engineering*. 305–308. https://doi.org/10.1109/FITME.2009.81

[188] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. 2020. Graph neural networks: A review of methods and applications. *AI Open* 1 (2020), 57–81. https://doi.org/10.1016/j.aiopen.2021.01.001

[189] Mingzhou Zhou, Xipeng Shen, Yaoqing Gao, and Graham Yiu. 2014. Space-efficient multi-versioning for input-adaptive feedback-driven program optimizations. *ACM SIGPLAN Notices* 49 (2014), 763 – 776. https://api.semanticscholar.org/CorpusID:1961378

[190] Eckart Zitzler, Marco Laumanns, and Lothar Thiele. 2001. SPEA2: Improving the strength pareto evolutionary algorithm. https://api.semanticscholar.org/CorpusID:16584254

[191] E. Zitzler and L. Thiele. 1999. Multiobjective evolutionary algorithms: a comparative case study and the strength Pareto approach. *IEEE Transactions on Evolutionary Computation* 3, 4 (1999), 257–271. https://doi.org/10.1109/4235.797969