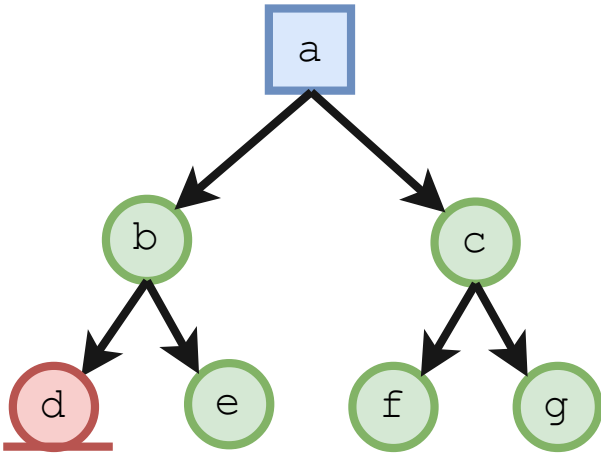
Imperial College London

Department of Electrical and Electronic Engineering

Final Year Project Report 2018 (Updated after Deadline)



Project Title:	**Inlining ML with ML**
Student:	**Fu Yong Quah**
CID:	**00921570**
Course:	**EIE4**
Project Supervisor:	**Dr David Thomas, Dr Andras Gyorgy**
Second Marker:	**Dr Thomas Clarke**

Abstract

OCaml, a strict functional programming (FP) language, has been gaining popularity in recent years, especially with the introduction of ReasonML (a dialect for web-programming) and MirageOS (a framework for building type-safe unikernels). Function inlining is an important optimisation in producing performant binaries in FP. Attention in machine learning (ML) guided inlining has mainly been given to Just-in-Time and low-level languages have primarily utilised low-level constructs such as types of machine instructions as features. This thesis presents a framework to study high-level function inlining coupled with a formally verified algorithm to label function call sites, an optimisation pipeline that generates data with iterative compilation, a novel demonstration-based reward assignment model for function call sites and a Mixture of Expert (MoE)-inspired model to make learn conservative classification decisions from noisy data labels.

When benchmarked against an optimising variant of the OCaml compiler, the pipeline produced a maximum speedup of 32.7%, in micro-benchmarks and a 1.93% median speedup in native compilation, when benchmarked against Flambda, an optimising variant of the OCaml compiler. In the compilation scenarios where the user wishes to benchmark different binaries produced via different compilation options, the pipeline provides the user with several speculative inlining policies. The median speedup of the generated binary is 5.88% and maximum speedup of 34.96%, compared to a median speedup of 3.02% and a maximum of 9.50% from a naive grid search of command line argument adjusting the compiler's inlining heuristics.

Acknowledgements

Throughout the course of the project and write up of the thesis, I have received invaluable support and guidance:

- *Dr. David Thomas* for supervising me throughout the project and inspiring me to take up function programming in lisp.
- *Dr. Andras Gyorgy* for supervising me in the machine learning components of the project, in addition to introducing me to the topic as my tutor.
- *My parents* for supporting me throughout my life.
- *My brother, Fu Xiang Quah* for regularly helping me decipher academic language and reminding me to uphold scientific integrity.
- *Nikolay Nikolov* for giving me suggestions of machine learning models to employ.
- *Alice Cao* for proof-reading initial drafts of this report.
- *Orsi Bojtar and Benjamin Withers (again)* for preparing (and largely simplifying) my otherwise incomprehensible and unnecessarily verbose presentation.
- *Andrew Li, Benjamin Withers, Bingjie Chan and Roy Lee* for being fantastic labmates throughout the duration of the project.

Contents

Glossary

apply_id An identifier to identify function call sites, guaranteed to be unique across the function's siblings regardless of any amount of function inlining.. 59

execution statistics A combination of execution time, speedup over baseline, garbage collection data and perf event counters of a program, usually coupled with an inlining tree denoting the set of taken inlining decisions.. 53, 61

expanded tree A expanded variant of inlining-tree, where all *Apply* and *Inline* nodes unrolled to expose the amount of function inlining that was required to represent it. . 8, 58, 59, 61, 62, 69, 72, 73, 98

Flambda An IR in the OCaml native compilation pipeline. High-level optimisations are performed in this IR.. 37, 41, 42, 74, 85, 86, 96, 108

hyperparameter a parameter that is not directly optimised by the system under analysis.. 61

incremental tree space exploration An iterative exploration problem.. 57

inlining heuristic An inlining heuristic is a form of inlining policy that employs human-designed features to decide whether to inline a function.. 57

inlining overrides The set of custom inlining decisions that a compiler user can pass to the compiler to override the compiler's default inlining heuristics decision.. 45

inlining policy An algorithm, policy or process employed by a compiler to decide whether to inline a function call.. 54

inlining trace Inlining trace refers to a path along the inlining trace from the root node. The inlining trace contains all the inlining decisions taken from the root up to any given decision.. 42, 43, 73

inlining tree A tree representation of the set of inlining decisions taken by a inlining compilation pass.. 8, 35, 50, 51, 58, 59, 72, 95

lambda code An IR in the OCaml native compilation pipeline. Lambda code is also used for bytecode interpretation.. 37, 42

maximal decision set of an inlining tree is the largest possible set of inlining decisions corresponding to the inlining tree. Every inlining tree has a unique maximal decision set.. 5, 43, 44, 101

minimal decision set of an inlining tree is the smallest possible set of inlining decisions corresponding to the inlining tree. Every inlining tree has a unique minimal decision set.. 43

parallel time The time complexity when there is an infinite number of processors available to process a task. For the purpose of benchmarking, the processors are assumed to be homogeneous.. 52, 54

profile guided optimisation An form of compiler optimisation that uses run-time data collected by running the program typically once, or at most a few times.. 57

reconstruction quality a metric that measures how well a predictive model captures the observed performance. 0 indicates the observed optimal performance, whereas 1 indicates the baseline performance. A lower value indicates better performance. Reconstruction quality is typically between 0 and 1, but it is entirely valid to lie outside this range.. 8, 65

reward assignment a process to assign numerical reward values inlining decisions at function call sites throughout a program via predictive modelling on demonstrations. This is described in chapter 6.. 73

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Objectives

The overall objective of this project is to investigate how machine learning can be used to help guide high-level inlining decisions in functional programming languages targeting native binaries. More concretely, the project aims to research ways where machine learning can be used to guide inlining decisions *for familiar programs* in a high-level intermediate representation (IR) in the OCaml optimising compiler[1], `ocamlopt`. The key objectives are:

- Design and implement a mathematical model to quantify the how inlining decisions in function call site throughout the program affects the overall program performance.
- Learning good inlining policies to perform function inlining based on static program features.

The goal of the project is not to create a general-purpose inlining compilation pass, but rather, deliver inlining passes targetted specifically for preoduction builds. It can be either a single inlining pass, or a set of N compilation passes, where the user benchmarks the N generated binaries. In other words, the target set of users of the project is not meant for normal development use, but rather, for users who care about program performance.

1.2 Challenges

- Targetting a native binary compilation means less feedback signal is generated. The most obvious signals available on in such use cases is the program execution time. Sampling tools (such as `perf`) gives minimal noisy details on the call graph and several coarse-grain details on the status of the processor.
- By operating on a high-level intermediate representations, instruction-level indicators simply are not readily available. For example: commonly used features in the literature such as number of memory accesses and number of conditional branches are not easily available longer. As a result, it becomes much harder to extract features from a given code snippet.
- From an engineering perspective, hacking a compiler is *hard*. Data structures are complex and sometimes subjected to human-imposed (and often undocumented) invariants[2].
- Running concurrent simulations is bounded by the amount of available hardware.
- Most compiler optimisations in procedural programming languages are oriented towards loops, such as: loop tiling, instruction-level parallelism, and loop nesting optimisation (Bacon

[1] The default OCaml optimising compiler performs almost no high-level optimisation at all. It is called an optimizing compiler as the other option is to use a byte-code compiler.

[2] An invariant is an informal agreement on the various fields of the data structure that is not indicated in the type system, often specified via comments. It is often desirable to have this being enforced by the type system or certain constructs, but the verbosity might result in higher engineering maintenance.

et al. 1994). In functional programming however, loops are specified using recursive function calls. The iterative nature of the code are indirectly represented as recursive function calls.

1.3 Structure

The report is structured as a thesis - the ideas and analysis discussed are not necessarily ordered chronologically. Chapter 2 discusses related background work and knowledge. Chapter 3 discusses and defines the scope of the project. The following chapters contain the main presentation and contributions of this project:

- chapter 4 presents a generalisable framework to investigate the study function inlining for programming languages.
- chapter 5 discusses iterative compilation techniques to generate data on inlining decisions at function call sites.
- chapter 6 presents a reward-assignment model that takes multiple runs of a same program with differing inlining decisions and assigns a numerical reward value to individual function call sites.
- chapter 7 discusses how uses machine learning can be used to derive an inlining policy.

Each of the chapters above contain sub-evaluation and discussion of the results, independent of other sections in the report. Chapter 8 provides a thorough analysis and evaluation of the derived inlining policy. The thesis in concluded with a in chapter 9 recapping the contribution of the project and relevant future work.

Chapter 2

Background

2.1 OCaml

OCaml is a strict functional programming developed, with roots from the Milner's Language (Milner 1997). As a functional programming language, it has a strong emphasis in the use of immutable data structures, first-class functions and abstractions. Immutable data structures refer to data structures that cannot be modified at runtime. An example of a commonly used immutable data structure is `List`.

OCaml ships with two compilers – a bytecode and a native compiler. The bytecode compiler, `ocamlc` generates platform-independent bytecode which can be executed by a OCaml bytecode intepreter, whereas the native compiler generates fast platform-specific native code. As intermodule information is exposed during compilation of OCaml source files, link-time Optimization (LTO) is less interesting in OCaml for high-level optimisation. This is as opposed to languages like C++ where different object files are compiled independently. This comes at the cost of a higher compilation overhead per file change. `ocamlopt`, which uses `gcc` for linking, is still able to leverage LTO to optimise dependence between its C object files.

2.1.1 Type System

Milner's type assignment system (Damas & Milner 1982) forms the foundation of OCaml's type system. While the exact formal reduction and assignment rules in the type system is not within the scope of the project, a basic understanding of the features provided by the type system will yield some ideas about locations where inlining opportunities can potentially arise. As in typed lambda calculus (**?**), OCaml's types have the following grammar:

$$\phi ::= \phi \to \phi \mid T$$

where $T \in$ base types.

Types are right associative, meaning $A \to B \to C$ is equivalent to $A \to (B \to C)$. This allows for currying when programming in OCaml, that is a function that takes multiple arguments can be translated to a function that takes a single argument with a return value of a function. For example: a function that takes two integer arguments and returns a string is equivalent to a function that take a single integer argument and returns a function that takes a single integer argument and returns a string.

Algebraic Data Types

Algebraic data types (ADTs) refer to composite types formed by combining other types. OCaml supports the two of the most common ADTs - product types and sum types. In OCaml, product

types are represented using tuples and records.

- **tuples** are a fixed-sized list of values, with length and types defined at compile-time [1]. The tuple type is formally defined as $(t_1 * t_2 * t_3 * ... * t_n)$, where t_i refers to an arbitrary type $\forall i$. Instances of a tuple exhibit the form $(a_1, a_2, a_3, ..., a_n)$ where $\forall i$. instanceof(a_i, t_i). Elements of a tuple can be accessed via pattern-matching or projection functions.

- **records** are a compile-time defined structures. Records are very similar to structs in C (not C++ – records in OCaml do not exhibit any of the Object Oriented Programming properties, nor support constructors). A record type is defined as $\{f_1 : t_1;\ f_2 : t_2;\ ...;\ f_n : t_n\}$, where t_i refers to an arbitrary type. An instance of a record type can be constructed with $\{f_1 = a_1;\ f_2 = a_2;\ ...;\ f_n = a_n\}$ where $\forall i$. $instanceof(a_i, t_i)$. Record fields can be accessed similar to structs in C – by using the dot operator, eg: `person.name`.

Sum types on the other hand, are represented using **variants**. Variants provide some form of polymorphism by by attaching. Variants as qualified union and enums. A variant type in OCaml can formally defined as $t_1 \mid t_2 \mid t_3 \mid ... \mid t_n$. OCaml provides variants in two different forms - standard variants and polymorphic variants. The difference in typing rules of polymorphic variants vs standard variants are akin comparing structural typing to nominative typing. Consequently, the flexibility offered from polymorphic variants implies a higher cost at runtime. For brevity, the typings rules (and hence formal definition) of product types, sum types, injection and reduction[2] are omitted.

Generic Parametric Types

Generic parametric types allows the programmer to parameterise types with generic type parameters – that is, a type that is composed of an arbitrary unknown type. A ubiquitous type that with such type parameters is the `list` type which allows storing an arbitrary number of elements of that given type. These type parameters are generic, meaning no assumptions can be made about those type parameters. Parametric type allows polymorphism (and reduces code duplication) in cases where no assumption about the type parameters is required. The generic type params requirement is as opposed to those in Haskell (Hudak et al. 1989) (Peterson et al. 1996) which allows type parameters to exhibit specific properties using typeclasses (Hall et al. 1996) (not too different from java interfaces). A code snippet that uses generic types with variants and records with generic type parameters is shown in figure 2.1.

2.1.2 Runtime Memory Representation and Garbage Collection

OCaml provides automatic memory management of a dedicated memory pool. This memory pool is referred to as the OCaml heap and is managed with a garbage collector. OCaml has a very simple runtime value representation – a word[3] is either an integer or a pointer to a block in the OCaml heap. The least significant bit of the word is a tag bit, 1 for integers and 0 for pointers to the OCaml heap. As a result, arithmetic operations on integers are less straightforward, requiring more machine instructions then a straightforward representation, but the cost is insignificant in modern processors (Minsky et al. 2013*a*).

Data structures in OCaml are represented in blocks with a simple uniform layout. Heap pointers of allocated memory are simply points to the beginning of a block[4]. Blocks, by default, are immutable, mostly due to OCaml's design as a functional programming language, but also in part due to the overhead incurred in write barriers (Fernandéz 2009).

[1] This is as opposed to tuples in python, where the length of tuples are not fixed at compile-time, but cannot be altered after creation. Python also does not enforce the types of elements within a tuple.

[2] Reduction rules are typing rules to transform a sum or product type into one of its underlying rules, whereas injection rules are for the converse.

[3] Word refers to a 32-bit value on a 32-bit machine and a 64-bit value on a 64-bit machine.

[4] Technically, the pointers points to $value_0$ because code typically do not require any information from the block headers, with the exception of dynamically sized data structures like arrays and strings.

```
type 'a option =
    | None
    | Some of 'a

type 'a t = {
     value  : 'a;
     left   : 'a t option;
     right  : 'a t option;
    }

(* [~f] is a named argument *)
let rec in_order_traversal : 'a t -> f:('a -> unit) -> unit  = fun tree ~f ->
  let descent_if_not_none t =
    match t with
    | None -> ()
    | Some t -> in_order_traversal t ~f
  in
  match tree with
  | Leaf a -> f a
  | Node node ->
    descent_if_not_none f node.left;
    f node.value;
    descent_if_not_none f node.right;
;;
```

Figure 2.1: A binary tree and a function to traverse it. Pattern match statements (`match ... with | ... -> ...`) are used to reduce variant types into its underlying types. Generic parameters are used for to allow `in_order_traversal` to work on binary trees with arbitrary underlying types.

block size	22 or 54 bits
GC color	2 bits
Tag byte	8 bits
$value_0$	1 word (32 or 64 bits)
$value_1$	1 word (32 or 64 bits)
$value_{2,3,\ldots,n-1}$	n - 2 words

Most types compiles to a similar block representation at runtime due to type erasure. The OCaml runtime does not contain (nor require) any information about the specific types of objects (as opposed to the Java Virtual Machine (JVM) or .NET runtime (CLR)). Nevertheless, there are a few notable special cases, such as:

- *Floating point and word-length integers*[5] - Due to the tag bit in arbitrary values, floating numbers cannot be simply be stored as a single word in memory. They are boxed in the above block format with a special tag byte denoting a floating point value. For similar reasons, word-length integers (32-bit or 64-bit integers) are boxed in the block format with a different special tag byte.
- *Function closures* - A block containing a "function closure" contains a combination of free variable assignments and function pointers. The specific structure [6] is unimportant for the purpose of this project.
- *Strings* - Strings are stored as an array of single-byte characters in the value fields, so characters do not consume a whole word. The values section for string blocks are also padded with zero-bytes (to which point it it is word-aligned), which enables C Foreign Function Interface (FFI) to use the value section directly as a C-string without unnecessary copying.

[5] All floating point numbers in OCaml are double precision on 64 bit machines

[6] The author has written a blog post to illustrate closure's structure, covering partial applications and currying, if it is of any interest to the reader.

The OCaml heap is dynamically resized when required. When a program runs out of memory, the runtime executes the Garbage Collector (GC) to free unused memory. GC in OCaml is generational (and also incremental, but the detail is less relevant here), as programming in OCaml (most ML-flavor dialect) involves allocating small amounts of memory at various places. The generational collector maintains two different memory regions within the OCaml heap (Minsky et al. 2013*b*):

- *Minor Heap* – a small fixed-sized memory region where blocks are initially allocated. Minor collection is performed using a copy collector, and live blocks are transferred to the major heap.
- *Major Heap* – a dynamically-resized memory region for blocks that have lived longer. GC in here is typically carried out using a mark and sweep collector.

Memory allocation in the minor heap is very fast (3 x86 instructions without garbage collection). Minor collection and major collection can contribute to the latency of a process, given that not much (directly) useful work is being done during GC cycles. Blocks are (usually) not allocated directly to the major heap – long-live blocks that survive a minor collection are automatically promoted to the major heap.

2.1.3 ML Modules and Functors

The simplest basic use of ML modules is to organise functions, values and types into separate groups. However, ML Modules can be used for more expressive use cases. Modules in OCaml are first-class objects with a separate namespace from the core language. In OCaml, modules that take a single module and return a new module are called functors (related, but not to be confused with functors in category theory or haskell). This is typically used to for code abstraction. Figure 2.2 shows a code snippet demonstrating the usage of functors for both the simple and expressive purpose. As modules can contain types, values and functions, modules can be used as a form of abstraction to generate composite types, but results in slightly more verbose code than implicit references in Haskell due to typeclasses. Modular Implicits (White et al. 2015) aim help rectify this verbosity and awkward programming style by introducing implicit module parameters , similar to implicit traits in Scala. As of writing, modular implicits are still not available in OCaml.

Under the hood, the runtime treats modules and functors similar to functions and values, in which it uses the same block structure as those of values. A functor simply returns a block whose fields contains pointers to closure blocks (for functions within modules) and regular values (integers or pointers to other forms of data structures). Having such a uniform runtime representation allows optimisation passes and garbage collections logic to be reusable over different constructs.

2.1.4 Compilation Pipeline

Compilation of a single OCaml module to assembly object files comprises of several stages. The flow chart in figure 2.3 shows the compilation pipeline for an OCaml source file. The figure shows the various intermediate representation (Intermediate Representation (IR)) the source code was processed in before generating machine code.

Compilation with `ocamlc` terminates right after generating lambda code. Consequently, compiling lambda code does not perform any form of aggressive optimisation. Bytecode is generated by linking .cmo files together. When producing native executables with `ocamlopt`, the compiler links the compiled object files of the modules with the OCaml runtime to produce a statically linked binary. Linking is performed using the system's gcc. The .cmx file produced is used by other modules that depends on the present module to import value approximations to enable cross module function inlining. The project investigates high-level optimisation in the FLambda IR (which is also the contents of `.cmx` files).

Rather confusingly, at every OCaml release, two variants of the OCaml compiler variants are distributed (each which each one copy of `ocamlc` and `ocamlopt`), one without Flambda (default) and another one with Flambda enabled. The generated object files by both compilers (both the Ocaml object file, `module.cmx` and the native object file, `module.o`) are not compatible. All OCaml source code that are linked together must be compiled with the same compiler variant.

```
module Person = struct
  type 'a t = {
     name : string;
     age  : int;
  }

  let compare a b =
    let c = String.compare a b in
    if c <> 0 then c
    else Int.compare a b
  ;;
end

module Make_map(module A : sig
                  type t

                  val compare : t -> t -> int
          end) = struct

  type 'b t = ...

  let find a : t -> 'b A.t -> 'b =
    ....
  ;;
end

module Person_map = Make_map(Person)
```

Figure 2.2: Simple example demonstrating the usage of functors and modules to generate a new type (ie: a person map type) that uses functors to create a new type (the Person Map type) and reuse the logic in . The analogy in haskell for this particular use case is typeclasses. The key difference is that in OCaml, the relevant functions that operate on the given type is explicitly given in the module argument, whereas in haskell, it is implicitly bind to the type argument.

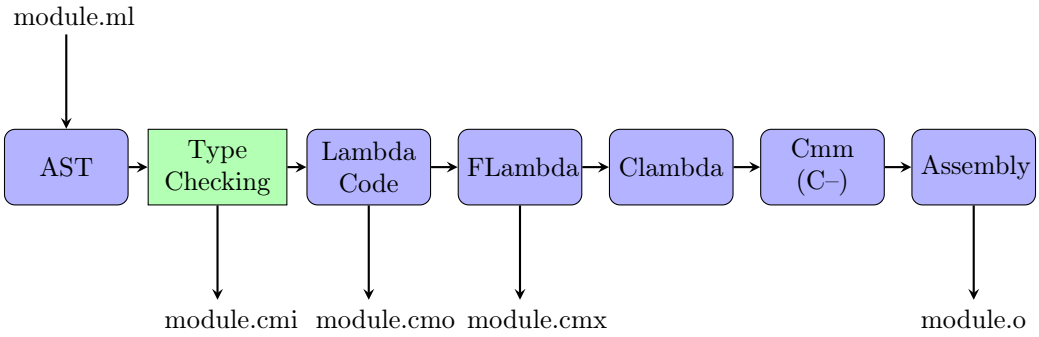


Figure 2.3: OCaml compilation pipeline. Rounded-corner boxes represents the intermediate representation form in which the OCaml compiler processes the program.

2.1.5 Flambda

Flambda is a series of optimisation passes provided by the native code compilers as of OCaml 4.03 that operates on the FLambda IR (Leroy et al. 2017). As opposed to the default compiler variant, which has a straightforward compilation strategy, the Flambda variant produces more optimised code. The Flambda IR is an tree-based IR, similar to an abstract syntax tree (AST). The Flambda compilation pass performs several different optimisation passes that operates solely on Flambda IR. These passes will be referred to ass 'Flambda passes' throughout the text.

The Flambda IR is an A Normal Form (ANF) (Sabry & Felleisen 1992). ANF allows the LHS of function applications to be lambdas or names. The Flambda IR has an additional restriction - that is the LHS term in function applications can only be referenced by name. In the Flambda IR, naming reference are down by symbols and variables[7]. Variables are used to refer to intermediate values during computation. Symbols are used only to reference values that are (1) constant; and (2) provided by another compilation unit or the current compilation unit's top-level module. The key types of values that can be represented by the Flambda IR includes the set of closures and function declarations. A set of closure is a set of function declarations, potentially mutually recursive. All function declarations are wrapped within a set of closures[8]. An exhaustive explaination about the types of values that can be represented Flambda IR are beyond the scope of the main text - interested readers can refer to appendix **??** for a more thorough explanation.

The available Flambda passes are (in not specific order):

- **Lift lets** increases the length of scopes to aid further optimisation. It flattens out let-bindings within nested let bindings to simplify value approximation at later passes.
- **Lift Constants** moves compile-time constants from the code body to symbols.
- **Share Constants** shares eligible constants that are shared to the top level. This constant applies for immutable objects (records, lists, tuples), but not constant reference to mutable values, such as strings, arrays or mutable blocks.
- **Remove unused program constructs** Remove unused symbols that have pure initialisation and Effect blocks that do not produce any side effects.
- **Lift let to initialise symbol** Transforms top-level lets in Flambda bodies
- **Remove unused closure variables** Remove unused function declarations in set of closures that are unused and not exposed to external modules.
- **Inline and simplify** Performs function specialising, function inlining and dead-code elimination *simultaneously*. This method walks the Flambda IR in a depth-first manner and inline functions where appropriate. Upon function inlining, code segments that are proved unreachable are removed as part of its optimisation procedure.

[7] In the actual Flambda implementation, for the sake of type safety, there exist different types for special naming references, such as `closure_id` for closures and `variable` for arbitrary variable.

[8] In OCaml compiler terminology, the terms closure and functions are often used interchangeably.

- **Ref to variables** Decompose a block with mutable fields into bindings to mutable variables.

2.2 Function Inlining

```
let f x =
  foo (x + 2 * 3)
in
f 123
```

```
let f x =
  foo (x + 2 * 3)
in
foo (123 + 2 * 3)  (* inlined [f] *)
```

Figure 2.4: An example of function inlining in OCaml. The right code snippet is a result of inlining `f` in the left code snippet.

To quote Jones & Marlow (2002), 'In principle, (function) inlining is dead simple, just replace the call of a function by an instance of its body'. An example of function inlining is shown in figure 2.4. Inlining was an optimisation first introduced as open compilation of subroutines (Cocke 1970). Whilst inlining is incredibly simple, it is often difficult to decide whether to inline a function. From a computational theory perspective, inlining can be viewed as a knapsack-problem (Scheifler 1977), where the code sizes are the weights of the tiems, and the inlined benefit as the of the value of the items. This illustrates that the problem is NP-Complete. On top of that, however, it is fairly ambigous what the inlining benefit can be - in scenarios where the target is to compile. Inlining is an important source of optimisation that permits a trade-off between code size and execution speed. Improvements in executed speed can be attributed to (1) direct effects due to the reduction in the number of instructions (Kaser & Ramakrishnan 1998); (2) Indirect effects due to cache eviction and virtual memory behaviour (Kaser & Ramakrishnan 1998); and (3) Compounded effects from other optimisation passes (Jagannathan & Wright 1996).

Over-agressive inlining, however, can lead to code bloat and degraded performance. The direct consequence of code-bloat is a overloaded and frequently evicted instruction cache and higher register-pressure at the call site. High register pressure is a result of an excessive amount of variables in body of functions when targetting processors (specifically, the ubiquitous x86_64 architecture) with very few registers. Register pressure results in register spilling during execution, offsetting any gains from inlining in the first place (Torvalds 2001). The fact that inlining a function can be a double-edge sword, computationally efficiently deciding the best set of inlining decisions for a given program is difficult. Ironically, Cooper et al. (1992) showed that unexpected performance loss from function inlining can be obvserved due to compounded effect from other compiler stages, such as compiler's instruction scheduler. In that specific work, the FORTRAN standard defines that address aliasing as undefined behaviour for performance reasons, but compilers handles aliasing at the expense of generating slower code. When functions are inlined, memory location aliasing is detected, resulting in the compiler generating stall instructions in critical loop iterations of floating point operations.

The default OCaml variant performs very limited rudimentary optimisations and inlining. It performs very a basic form of inlining, that simply checks the size of the function body against a pre-determined threshold and does not run multiple optimisation passes over the IR. As a matter of fact, whether or not a function is inlined is pre-determined *during* the function declaration. The simple compilation strategy results in a relatively straightforward mapping from source code to machine instructions.

2.3 Machine Learning

Machine learning is a field in computer science that uses computers the ability to learn based on past data without being explicitly programmed to make certain decisions (Samuel 1959). The 3 main classes of machine learning includes (1) supervised learning, where the dataset comprises of pairs of observations and associated labels (2) unsupervised learning, where the dataset comprises of just observations and (3) reinfocerment learning, which will be explained in a later subsection. Machine learning problems are typically expressed as an optimisation problem, learn a set of

values (commonly referred to as *weights*) that minimises / maximise a given objective $\mathbf{w}^* = \arg\min_{\mathbf{w}} J(\mathbf{w} \mid \mathcal{D})$ given a dataset $\mathcal{D}$. Analytical solutions for these minimisation objectives typically do not exist, so they are usually minimised using gradient-based iterative solutions, such as stochastic gradient descent (Bottou 2010).

2.3.1 Classification

Classification is a task where a new observation is assigned a category, $\mathcal{C}$ based on training set containing observations whereby the finite set of categories of past observations are known. Classification is a form of supervised learning, where the label is the category of the observation. Classification algorithms generally operate on observations are finite vectors, namely $\mathbf{x} \in \mathbb{R}^N$ [9], whereby it learns a function $f : \mathbb{R}^N \to [0, 1]^{|\mathcal{C}|}$

Multinomial Logistic Regression

Multinomial logistic regression (Greene 2003) (also known as softmax regression) is a classification algorithm that generalises logistic regression (Walker & Duncan 1967) to multiple classes. It determines the probability of an observation in a given category by using a linear combination of features:

$$P(\hat{\mathbf{y}} = c) = \frac{\exp(\mathbf{w_c}^T \mathbf{x})}{\sum_{k \in \mathcal{C}} \exp(\mathbf{w_k}^T \mathbf{x})} \tag{2.1}$$

The minimisation objective is the cross-entropy loss (also known as the negative log-likelihood loss):

$$J(\mathbf{W}) = -\sum_{c \in \mathcal{C}} \sum_{i \in c} \log \frac{\exp(\mathbf{w_c}^T \mathbf{x}^{(i)})}{\sum_{k \in \mathcal{C}} \exp(\mathbf{w_k}^T \mathbf{x}^{(i)})} \tag{2.2}$$

where $\mathbf{w_c}$ refers to the c-th column of the matrix $\mathbf{W} \in \mathbb{R}^{|\mathcal{C}| \times N}$.

An analytical solution for logistic regression is not known to exist and hence, is trained iteratively using stochastic gradient descent.

Linear Discriminant Analysis (LDA)

LDA (Fisher 1936) is a classification algorithm that classifies objects into $\mathcal{C}$ based on a linear combination of features. LDA was originally presented as a classification algorithm for two classes, a multiclass generalisation was presented by Rao (1948). Specifically, LDA learns a linear projection from $\mathbb{R}^N$ to $\mathbb{R}^{|\mathcal{C}|-1}$ and uses bayes rules to predict the class probability. LDA assumes that data points for a given class forms a gaussian distribution and gaussians for each categories' probability distribution shares the same covariance matrix $\forall k \; . \; \Sigma_k = \Sigma$.

The maximisation objective of multi-class LDA is defined as follows:

$$J(\mathbf{W}) = \frac{|\mathbf{W}^T S_B \mathbf{W}|}{|\mathbf{W}^T S_W \mathbf{W}|} \tag{2.3}$$

$$S_B = \sum_{c \in \mathcal{C}} (\mu_c - \bar{x})(\mu_c - \bar{x})^T \tag{2.4}$$

$$S_W = \sum_{c \in \mathcal{C}} \sum_{i \in c} (x_i - \mu_c)(x_i - \mu_c)^T \tag{2.5}$$

[9] Models that operate on sequences exist, such as LSTM presented by Gers et al. (1999), but are beyond the scope of this project

where S_B defines the separation between classes, and S_W defines the separation within a class. Intuitively, the optimisation objective of LDA is to learn a linear projection that balances between maximising the separation of distribution between classes and minimising the separation of distribution within a class. A solution for LDA exists, when S_W is a full rank matrix (Tharwat et al. 2017).

Artificial Neural Networks

Artificial neural networks (ANN) (McCulloch & Pitts 1943) is a computing system inspired from biological neuron connections. In the context of classification, ANNs are used to assign class probabilities to observations based on a non-linear combination of features.

A fully-connected ANN for classification to $\mathcal{C}$ is typically defined by the parameters of the hidden layers $(h_1, a_1, h_2, a_2,$
$..., h_{n_{hidden}}, a_{n_{hidden}}$. h_i describes the number of hidden layers in layer i whereas a_i describes the activation function of hidden layer i. Classification in ANN is performed as follows (with $f(\mathbf{x}) = f_1(\mathbf{x})$):

$$f_k(\mathbf{x}) = \begin{cases} f_{k+1}(a_k(W^{(k)}\mathbf{x})) & k <= n_{hidden} \\ a_{output}(W^{(o)}\mathbf{x}) & k = n_{hidden} + 1 \end{cases}$$

$$a_{output}(\mathbf{z}) = \frac{\exp(\mathbf{z})}{\sum_{k \in \mathcal{C}} \exp(z_k)} \tag{2.7}$$

where $W^{(1)} \in \mathbb{R}^{h_1 \times N}$, $W^{(k)} \in \mathbb{R}^{h_k \times h_{k-1}}$ and $W^{(o)} \in \mathbb{R}^{|\mathcal{C}| \times h_{n_{hidden}}}$. Passing a feature vector $\mathbf{x}$ into a ANN returns a vector denoting the probability distribution over the classification categories. Similar to multinomial logistic regression, the optimisation objective, defined in terms of the matrix, for classification ANN is defined by the cross-entropy loss in the dataset.

$$J(\mathbf{W}^{(1)}, \mathbf{W}^{(2)}, ..., \mathbf{W}^{(n_{hidden})}, \mathbf{W}^{(o)}) = -\sum_{c \in \mathcal{C}} \sum_{i \in c} \log \frac{\exp(f(\mathbf{x}^{(i)}))}{\sum_{k \in \mathcal{C}} \exp(f(\mathbf{x}^{(i)}))} \tag{2.8}$$

The main benefit of using an ANN is that it can typically learn fairly arbitrary non-linear combination of features suited for the classification task on hand. A primary downside of using neural network is the increased risk of overfitting the machine learning model towards the training set, resulting in an ungeneralisable solution. As there is no analytical solution for an arbitrary neural network, hence training is usually performed using gradient-based method with backpropogation (Rumelhart et al. 1986).

2.3.2 Clustering

Clustering is a form of machine learning that takes assign groups to objects such that members of a group have similar characteristics Bailey (1994). Clustering is an unsupervised learning task, as it simply operates on the properties of the objects without their labels. The similarity between objects are measured using a distance metric. Similar objects have a low distance measure. A commonly used distance metric for finite vectors is the euclidean distance, $d(\mathbf{u}, \mathbf{v}) = (\mathbf{u} - \mathbf{v})^T(\mathbf{u} - \mathbf{v})$.

K-Means Clustering

K-means clustering is a centroid-based clustering algorithm, whereby clusters are represented by a single centroid vector. This method assigns feature vectors $\mathbf{x} \in \mathbb{R}^N$ to one of K possible clusters. A common algorithm to perform K-means clustering is the iterative Lloyds's algorithm, which proceeds in two steps (MacKay & Mac Kay 2003):

- **Assignment** $C^{(t)}(x_i) = \arg\min_{k \in K} ||x_i - m_k^{(t)}||^2$. The set of data points that are assigned to a cluster k is defined as $S_k^{(t)} = \{x_i \mid C^{(t)}(x_i) = k\}$
- **Update** The new means of the clusters are updated to the centroids of all members of the cluster $\mathbf{m}_k^{(t+1)} = \frac{1}{|S_k^{(t)}|} \sum_{\mathbf{x} \in S_k^{(t)}} \mathbf{x}$.

Depending on the choice of initial centroids $\mathbf{m}_k^{(0)}$, the algorithm may not necessarily converge to an optimum (Hartigan & Wong 1979). A common simple initialisation step is the Forgy partition method (Hamerly & Elkan 2002), where K randomly selected points from the dataset as initial centroids. There exist various other methods, such as kmeans++ (Arthur & Vassilvitskii 2007), that provides a better centroid initialisation. New unseen observations is assigned a cluster by choosing the cluster centroid with the lowest euclidean distance.

2.3.3 Dimensionality Reduction

Dimensionality reduction is a form of machine learning that takes a training set learns a function that projects a training set of $\mathbb{R}^N$ to $\mathbb{R}^D$ where $N \gg D$. Dimensionality reduction is typically a form of unsupervised learning, whereby labels on individual observations are not required, but there exists several form of dimensionality reductions that makes use of the labels of each datapoint. Dimensionality reduction is useful for (1) mitigating the effects of curse of dimensionality when applying clustering or nearest-neighbour algorithms (Beyer et al. 1999); and (2) project the features in 2D or 3D space so that it can easily be visualised.

Principle Component Analysis (PCA)

Principle Component Analysis (PCA) (KPFRS 1901, Hotelling 1933, 1936) is a statistical technique that learns a projection matrix to map a training set with potentially linearly correlated dataset into one with orthogonal features. One way to find the projection matrix is to use . Given a training dataset $X \in \mathbb{R}^{M \times N}$, it learns a projection matrix $W \in \mathbb{R}^{N \times N}$, whose columns are composed by the eigenvectors of to covariance matrix $\frac{1}{M} X^T X \in \mathbb{R}^{N \times N}$. Upon learning projection matrix, the a new observation (feature vector) can be projected into the new feature space by $f(\mathbf{x}) = W^T \mathbf{x}$. Eigenvectors with larger eigenvalues corresponds to feature combinations that explains higher variation in the dataset. A projection matrix that reduces the number of dimensions can be attained by taking the eigenvectors with the biggest m eigenvalues to construct a dimentionality-reduction projection matrix U. PCA is useful for dimensionality reduction whilst retaining linear variation in the dataset.

Linear Discriminant Analysis (LDA)

On top of being used for classification, LDA can be utilise as a form of linear dimensionality reduction that projects the training data to $|\mathcal{C}| - 1$ components. As LDA linearly projects the data that maximises the separation between the centroids of data belonging to different categories, it helps to visualise the separability between categories of data. It is sometimes use as a preprocessing step before running other forms of classification algorithms (such as ANN or nearest neighbours).

2.3.4 Reinforcement Learning (RL)

Reinforcement learning (RL) is learning how to map situations to actions with the goal of maximising the total numerical reward signal received (Sutton & Barto 1998). There is no explicit

"learning" and "inference" stage - an agent is placed in an environment and has to figure out what to do. Reinforcement learning problems are commonly expressed in terms of a Markov Decision Process (MDP), which is defined by $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$ where the goal of an agent in RL is to learn a policy that maximises the long-term reward. The following definitions are a paraphrase of definitions due to Sutton & Barto (1998):

- $\mathcal{S}$ is a finite set of states
- $\mathcal{A}$ is a finite set of actions
- $\mathcal{P}$ is the state transition probability, ie: $\forall\,(s, a, s') \in (S \times A \times S)\;.\; P(S_{t+1} = s' \mid S_t = s, A_t = a)$
- $\mathcal{R}$ is the reward function, $\forall (s, a)\;.\; E[R_{t_1} \mid S_t = s, A_t = a]$
- A *policy* defines how an agent chooses an action at a given state. A policy is usually defined as a probability distribution, namely $\forall (s, a) \in (S \times A)\;.\; P^{\pi}(a \mid s_t = s)$
- *Long Term Reward* at a given time step is defined as, $E^{\pi}[R_{t+1}|S_t = s] = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$ where γ is the discount factor

These definitions naturally yields the state-value function: $v_{\pi}(s) = E[R_{t+1} + \gamma v_{\pi}(S_{t+1}) \mid S_t = s]$ The holy grail of an agent is to learn the optimal policy, that has a state-value function of $v_{\pi^*}(s) = \max_{\pi} v_{\pi}(s)$. There exist policies that uses the value function as part of its decision process. RL spans a large scope of problems and research areas.

Dynamic Programming (DP)

Dynamic Programming (DP) can be used to found the optimal policy when the model[10] of a the problem is known. Two common dynamic programming algorithms are value iteration and policy iteration (Kaelbling et al. 1996). DP solutions to problems are generally feasible only if the state space is computationally feasible to iterate within a given time budget.

Inverse Reinforcement Learning (IRL)

Russell (1998) characterises the IRL problem as follows: Given (1) measurements of the agent's behaviour over a variety of circumstances; (2) measurement of sensory inputs to the agent; and (3) if available, the model of the environment, determine the *reward function* being optimised. As opposed to standard RL, which learns an optimal policy π, the objective of IRL is to learn reward $\mathcal{R}$ for state-action pairs $S \times A$. Ng et al. (2000) presents a LP formulation for solving $\mathcal{R}$ given $\mathcal{S}$, $\mathcal{A}$, $\mathcal{P}$, γ and an optimal policy π^* in finite space spaces and using a linear function approximator as the reward function in large state spaces. Christiano et al. (2017) presents an optimisation flow that iteratively learns an estimate of the reward function using supervised learning by comparing human preference for different state-action trajectories.

2.4 Related Work

2.4.1 Inlining in Functional Programming

Jagannathan & Wright (1996) proposes Flow-Directed Inlining that uses the control-flow analysis to identify candidate call sites in functional programming languages, like Scheme and ML. To quote Shivers (1991), 'Flow analysis for a high-order language determines the set of values to which variables may be bound and the sets of values that expressions may yield'. The key idea presented in this work is to perform analysis on the control flow graph (CFG) and consider call sites as inlining candidates only if their arguments' values are known within the closure, as those methods would lead to further function function specialisation opportunities. This work also makes a (then) unorthodox claim in the literature that function size is not a good indicator for inlining

[10] Model in RL refers to the probability transition and reward function

```
let x = foo 100 in
x + x
```

```
let x = foo 100 in
foo 100 + foo 100
```

Figure 2.5: Expression inlining that inlines both occurrence of x. Work duplication as a result of excessive expression inlining in haskell.

heuristic, as more often than not, large conditional statements can be eliminated with dead-code elimination when the specific values of function arguments are known.

Serrano (1997) proposed experimental measures that have driven the design of inline function expansion in the context of the Scheme programming language, namely *when* and *how* to do inlining. It propose an algorithm primarily based on simple rules: (1) body-size vs call-size ; and (2) if the body of `f` after inlining will shrink by further optimisation to become smaller than the overhead of function call. The authors referenced Jagannathan & Wright (1996), noting that the two work are complementary - indeed code size reduction can be achieved using flow analysis to determine unreachable code branches during compilation. The relevance of the work in Scheme programming language to those in OCaml is mainly in the context of sum types. Inline expansion reduces resulting code size.

The *Glasgow Haskell Compiler (GHC)* (Jones et al. 1993), one of the most popular compiler for the haskell programming language, uses inlining heavily for optimisation passes over GHC Core (Jones & Marlow 2002), an IR in GHC. This is one of the first formal work that thoroughly documents the tricks and techniques in making a good inliner in functional programming languages. Inlining in haskell spans more than just function inlining. As haskell is a lazy and pure functional programming language, GHC supports inline arbitrary expressions in variable bindings, running the risk of duplicating work. One of the inlining heuristics in haskell is that it does not duplicate work, depicited in an example shown in figure 2.5. Due to the ability to inline arbitrary expressions, function inlining is achieved by two steps in GHC, (1) inlining the LHS of the function call site with the said function; and (2) eliminate the call with β-reduction. The shown example of inlining essentially destroys the work done by Common Sub-expression elimination (CSE) optimisation. Several key ideas with regards to inlining heuristic in GHC are (1) GHC selects a loop breaker (of which it never inlines) in a set of mutually recursive functions to ensure termination; and (2) GHC inlines 'small-enough' functions, use a set of hand-tuned heuristics to decide whether to inline big-size functions and do not consider inlining functions larger than a certain size to prevent code bloat (GHC 2015). As a consequence of (1), GHC *never inlines* any recursive functions, even those with static arguments.

Flambda performs various kinds of optimising, such as lambda-lifting, closure unboxing and function inlining. One of the key difference between Flambda and GHC's inliner is that Flambda inlines recursive functions, exposing more opportunities for loop unrolling (although the functions are not explicitly annotated as loops). Flambda's inlining heuristic is a linear combination of hand-designed function features. For efficiency reasons, Flambda first computes a coarse estimate of the inlining benefit before computing a more fine-grained benefit, both of which are linear combinations of hand-designed features. The Flambda inlining heuristics are computed from the estimated benefit from inlining (Chambart et al. 2017), composed from a linear combination of the following features:

- **Removed function calls**
- **Remove heap allocations** - As of writing, the compiler does not account for the amount of words being allocated, which are sometimes statically known.
- **Removed primitive operations** Primitive op refers to Lambda code operation. Some lambda code operations are potentially expensive such as writing to a block, which incurs the penalty from write barriers. As of writing, Flambda does not assign different weights for different operations.
- **Removed code branches** - Code branches in the inlined function body can sometimes be proven to be unreachable, and hence, removed by dead code elimination (DCE).
- **Direct call of indirect** - number of indirect function calls that are transformed into direct function calls. An direct function call is akin a function call by pointer, whereas a direct

function call is one where the callee is statically known.

Flambda then inlines the function if $size_{inline} - P(call) \times benefit \leq size_{original}$, where $P(call)$ is calculated as $\frac{1}{(1+branch\_factor)} branch\_depth$. Reasonable weights are shipped with the compiler, although the user can specify custom weights as the above features in command line flags.

2.4.2 Machine Learning in Compiler Optimisation

Agakov et al. (2006) uses machine learning for to constraint the search space in iterative compilation, namely, it optimises the sequence of compiler transformations. The work first builds a independent identically distributed (IID) model of probabilities that program transformation being "good" [11] (defined as yielding at least 95% of the maximum performance), that is $P(s_1, s_2, s_3, ..., s_L) = \prod_{i=1}^{L} P(s_i)$. The authors also describe a Markov model for program transformations, that is defining the probability of program transformations being "good" as $P(s_1, s_2, s_3, ..., s_L) = P(s_1) \prod_{i=2}^{L} P(s_i|s_{i-1})$. Both the IID and markov models are constructed for 12 representative benchmarks. Machine learning used for (1) *Feature Reduction* - Every compilation target (ie: program) is characterised by 36 hand-picked loop-level features. These features are reduced using Principal Component analysis (PCA) to 5 features, and (2) *classification* - Nearest Neighbour classification is used to compare the target program with respect to the 12 benchmarks where the IID and Markov oracles are tuned against. The authors report that iterative compilation converges to an optimal solution than those of random evaluation, by running C++ benchmarks on AMD, after 10 evaluations, a Random walk achieves 22% of the maximum performance, whereas the IID model and markove model achieve 41% and 66% respectively. Similar improvement was observed in a randomized genetic algorithm (GA) search versus a guided GA search.

Coons et al. (2008) uses machine learning to optimise instruction placement in a Explicit Dataflow Graph Execution (EDGE) processor. The compiler of an EDGE processor must manage both concurrency and communication at a fine granularity. EDGE system compiles source code into statically-allocated hyperblocks with many instructions; the hyperblocks are dynamically scheduled by the CPU (contributors 2017). Using a set of features for an instruction placement decision, the authors curate a dataset of features versus execution time on hardware by systematically varying the coefficients of every feature. Poor performance predictors are removed by lasso regression (Tibshirani 1996) whereas redundant features are eliminated with traditional correlation techniques. Using the remaining (small) set of features, the author then goes on to train a cost function that finds good placement. The cost function is trained using Neural Evolution Augmenting Topology (NEAT)(Stanley & Miikkulainen 2002), a reinforcement learning technique to evolve neural networks (McCulloch & Pitts 1943). Interestingly, the authors state that 'the hand-tuned heuristics described in prior work are actually very good general solutions; the heuristics learned on a training set of benchmarks were unable to significantly improve upon them when tested on novel benchmarks', but 'allow(-ing) the compiler to choose from among a variety of heuristics based on characteristics of code being compiled' is effective. Hence, they have performed clustering on segments of code and trained a placement heuristic cost function per cluster and goes on to suggest that similar methods ought to be explored in the domain of function inlining, register allocation and heuristics-guided decisions in compilers.

A common problem in applying machine learning to compiler optimisation is the sparsity of benchmarks, making it unsuitable for machine learning techniques. Cummins et al. (2017*b*) presents a method for synthesising benchmarks using a LSTM-based character model to generate openCL benchmarks, *CLGen*. The usage of large number of benchmarks generated by *CLGen* helps improves on previous work's predictive model's generated performance and aids in the discovery of newer source features for performance improvements. Cummins et al. (2017*a*) further improves on this work by using deep learning to automatically construct further compiler heuristics for GPU thread coarsening factors using raw code. The problem with regards to sparsity is once again

[11] To the layman reader – why would there ever be a compilation pass that deliberately degrades the performance compilers? Jones & Marlow (2002) states that some compilation passes deliberately degrade the program to expose the program to further optimisation opportunities to other passes. This is not true of all optimisation frameworks – Flambda tries to do only transformation that improves a program. (No citation available - I am pretty certain this is the case, having worked with the core Flambda developers and contributed to its codebase itself)

presented by Wang & O'Boyle (2018) in a review paper on machine learning for compiler optimisations, further adding that the empirical benchmarks must be a representative of the design space of user-written programs (legal programs are not necessary a representative of a sample from the distribution of programs).

2.4.3 Machine Learning in Function Inlining

Cavazos & O'Boyle (2005) uses generic algorithm to tune 5 inlining heuristic parameters, namely the calle max size, always inline size, max inline depth, caller max size and hot callee max size in the Jikes RVM, a research variant of the Java Virtual Machine (JVM). The parameter exploration is conducted a representative set of benchmarks. The evaluated parameters are meant to be suited for general purpose use, surpassing the performance of the default hand-picked parameters. Further performance gains were obtained when heuristics' parameters are tuned specifically for a benchmark, making it a form of iterative compilation.

Lopes (2010) to make function inlining decisions in LLVM (Lattner & Adve 2004) bit code by training a heuristic function using AdaBoost. The author takes an interesting approach towards generating classification data. To generate classification data for a single benchmark, firstly, it takes the target program / benchmark and compile it to a single LLVM bitcode file. The baseline execution time is measured by compiling the bitcode file with `-O2`, except that it disables the function inlining LLVM pass. Following the baseline, many independent inlining trials are carried out, whereby a different function call site is inlined in each one of them. Every function call sites' are then labelled based on whether their respective inlining trials produce a speedup over the baseline (where no function calls are inlined). The labelled data over function call sites over a set of benchmarks are then used to train an AdaBoost classifier. The offline approach towards studying inlining is a novel, but the work does not give care to study the relationship between different inlining decisions and assumes complete independence between inlining decisions for classification.

Kulkarni et al. (2013) proposes the use of micro-evolution to learn good inlining heuristics using NEAT in a highly tuned Java HotSpot server. As it is difficult to pin the impact of a single inlining decision in isolation, NEAT is used to iteratively evolve the inlining policy. By evolving the inlining policy on representative benchmarks, it learns a general purpose inlining policy. The work uses low-level caller, callee and call context features such as number of number of branch instructions, number of method calls, whether or not the program is in a loop, whether or not the program is a synchronized method[12]. The author also propose a method for generating human-readable heuristics, in which NEAT is used to label function call sites from the training data to train a decision tree.

To the author's best knowledge, there has not be thorough work on evaluating machine-learning based function inlining for high-level IRs and typed functional programming languages. There is no definite framework that preserves the relationship between function calls to study function inlining (machine learning work about simply assumes independence between function calls, or looks at shallow function calls), nor is there any work that frames function inlining as an offline optimisation problem.

[12] A java only artifact that limits only one subject from calling an object method at a point in time

Chapter 3

Problem Specification and Analysis

The goal of this chapter is to define the questions this thesis tries to answer and define a realistic project scope. Firstly, the nature of function inlining on the impact of performance of OCaml programs are is presented, whilst presenting some suggestive indicator for inlining opportunities. Then, the mathematical and engineering challenges in studying function inlining will be discussed, in the context of both Flambda for OCaml and general-purpose programs. At the end of this chapter, a problem definition is given and an overview of the solution proposed by this thesis is presented.

NB: In this chapter and the text that follows, the term "function" corresponds to those in the IR, which means it could refer to either a functor or a closure (ie: a normal OCaml function). The terminologies "faster" and "performance increase" will be used interchangeably.

3.1 Performance in OCaml Programs

In the background, an introduction to the OCaml runtime, type system and language features has been presented. Using this information, along with intuition developed from writing OCaml programs, a set of hypothesis describing the relationship between making inlining decisions and the program's performance can be formulated.

3.1.1 OCaml Runtime

As discussed in the background section, the OCaml runtime uses a tag bit to indicates whether a machine word is an immediate integer or a pointer to a block on the OCaml heap. As a result of this structure, IEEE-754 floating point numbers need to be boxed 2.1.2. Calling a function that takes a floating point argument might impose an addition function allocation, as illustrated in the example in figure 3.1.1. In the example, (assuming `print_float` does not result in any allocation) if `f` is not inlined, every loop iteration will result in two block allocations, 1 to wrap the floating point argument to function `f`, and another to wrap its return value. As of writing, neither Flambda nor the closure compiler gives special treatment to floating point. This gives rise to the a hypothesis:

Hypothesis 3.1.1. *Functions with floating point arguments and return values ought to be inlined more aggressively based on its substructure.*

The main goal of hypothesis 3.1.1 is to illustrate that certain properties of functions implies that values will be allocated implicitly without an explicit block construction. A more generalised hypothesis describing this is:

Hypothesis 3.1.2. *Function arguments, body and return values should be analysed for implicit value allocations, on top of explicit block construction in the function body.*

An example whereby an implicit allocation will be imposed in the absence of function inlining is shown in figure 3.1.1. By inlining `foo`, the let binding that creates a closure block to wrap `f` is no

```
let f x =
   1 +. x
;;

let () =
  let arr = [| 1;0; 2.0; 3.0; |] in
  Array.iter (fun x ->
      print_float (f x))
    arr
;;
```

Figure 3.1: A simple function where inlining `f` will significantly reduce the number of allocations and increase performance.

longer needed, and hence, will be removed in a later compilation pass. This allocation-removal is implicit, as it is not immediately obvious from the function body of `foo` alone.

```
let foo f x =
   f (x * 2 + 1)
;;

let bar env =
  let f x =
    env * x
  in
  foo f 123
;;
```

Figure 3.2: When `foo` is not inlined in `bar`, a value is implicitly allocated to construct a closure block for wrapping `f`.

A fundamental assumption of the OCaml garbage collector is that ML code tends to produce many short-lived and small allocations. Hence, the minor heap is small and is collected with a fast copy collector. Reachable memory segments are then promoted to the major heap, which is garbage-collected with a slower mark-and-sweep collector. In fancier terms, not all allocations ought to be treated the same:

Hypothesis 3.1.3. *Minor collections do not result in a significant impact in performance as opposed to major collections. When weighting how GC affects performance, the primary indicator ought to be major collections.*

Trying to thoroughly evaluate this hypothesis itself is an open-ended problem. A proxy hypothesis would be to argue that function inlining that removes long-lived global value allocations should be given a higher weightage than short-term local ones.

3.1.2 Programming Abstractions

Type parameters and functional programming gives rise to very generic programming and abstraction and work has been put into ensuring that generics perform as well as non-generic counterparts. such as those by Magalhaes et al. (2010) for GHC. All things equal[1], it is possible that inlining a generic function, can result in a more massive performance. The rationale for this is that the programme, have had to make certain abstractions in its argument so that the function have a generic type signature. 3.1.2 shows a code-listing where inlining will potentially result in massive performance improvements. When `foo` is inlined, it can make specific assumptions about the types of the `'a` type referenced by `a` and `f` in the inlined variant.

[1]Two functions with the same body in Flambda IR need not have the same type signature, due to OCaml's generic block representation for values. This can also happen (especially in the case of generics) in the presence of high order functions

```
(** returns [f a; f (f a); f (f (f a)); ...] **)
let rec foo ~f a n : ('a -> ('a -> 'a) -> int -> 'a list) =
  let loop ~prev ~iter ~acc =
    if iter = n then
      List.rev acc
    else
      let cur = f prev in
      let acc = cur :: acc in
      loop ~prev:cur ~iter:(iter + 1) ~acc
    in
  loop ~prev:a ~iter:0 ~acc:[]
;;
```

Figure 3.3: A generic function with high inlining potential.

Hypothesis 3.1.4. *OCaml functions take take argument with generic parameters, especially those with types with arrows (eg: `'a -> 'b`), ought to be considered for inlining more aggressively than non-generic counterparts.*

Working with kinds (ie: type of types) is almost a prerequisite of ML-like languages that provides higher-order type systems. It allows code reusability without the type system coming in the way. Consequently, on top of hypothesis 3.1.4, it is possible to go a step further and make a bolder claim for the case with functors:

Hypothesis 3.1.5. *OCaml functors that take type arguments and returns a new modules with new types can provide interesting avenue for function inlining ought to be inlined more aggressively than similar counterparts.*

One might comment that inlining a type-constructing functor all the time is generally a bad idea due to code-bloat. Consider `Map.Make`[2], a stdlib functor that generates a BST-based immutable lookup table. This belongs to classes a functor whom bodies are extremely big. However, at a closer observation of the functor contents of the `Map.Make` functor: (1) A key benefit from inlining function calls in high-level IRs is exposing further optimisation opportunities (Jagannathan & Wright 1996). If a functor is not inlined, many type-specialised function optimisations cannot be attained; (2) many of its functions have arguments whose types are generic, eg: `find`, `map` and `iter`. Those code themselves might be further candidates for inlining - following the hypothesis 3.1.4, these code is perfect for inlining; and (3) It contains *a lot* of recursive functions, many of which are tail-call recursive functions. These are avenues that provides very aggressive inlining opportunities. The benefits that are expected from inlining functors that return types can be compared to those from inlining template functions in C++.

3.2 Challenges in Studying Inlining

3.2.1 Problem Formulation Challenges

The role of function inlining in the context of performance is simple - make a set of inlining decisions *globally across the entire program* to yield a faster binary. As shown in the background, formulating inlining as a global optimisation can be approximately reduced to a NP-Complete problem (Scheifler 1977). The approximation fails to appreciate that the benefits obtained from inlining different functions are not independent on one another.

When consider sibling function calls sites in an AST, there is no guarantee that the optimal inlining decisions are independent of one another. The right amount of inlining will result in code that can fit in the processor's I-cache. Such inlining decisions need to be made in parallel, breaking the independence assumption.

[2] https://github.com/ocaml/ocaml/blob/4ea470feb10e2e1a85c2565d640a47fe9dc9a8e3/stdlib/map.ml

```
type t =
  | A of int
  | B of string
  | C of float

let foo ~f a =
   printf "a function call\n";
   f a
;;

let s =
  foo ~f:(fun x ->
      match x with
      | A x -> string_of_int x
      | B s -> s
      | C x -> string_of_float x)
    (C 1.0)
in
printf "%s" s
```

Figure 3.4: An example of non obvious inlining opportunities. Two levels of inlining is required to obtain substantial performance improvements.

Besides that, dependence can also arise from inlining decisions in nodes that are distance-apart in the AST. Given a function f and a call site s, the choice of inlining decision at call site s is dependent set of inlining decisions made when analysing the body of function f. Obviously, it is better to inline a function which has undergone minimal inlining at declaration, as opposed to one whose body have been thoroughly inlined.

On top of the problem of assuming independence between inlining decisions, *the relation between short term and long term benefits* poses a dilemma in designing inlining heuristics. The Flambda pass in OCaml mainly accounts for short-term and immediate benefits (Chambart et al. 2017), such as block allocations as a result of function inlining. The upside for doing this is a simpler compiler and comprehensible inlining heuristic. As a matter of fact, the OCaml compiler allows the user to customise the weights of its inlining heuristic. However, for any realistic optimisation opportunities, the possibilities for making more speculative inlining decisions ought to be considered, on the basis that there will be further decisions down the line, as shown in the example in figure 3.4. In the given example, inlining `foo` will unlikely to result in speedup, but inlining `f` upon inlining `foo` will result in substantial dead-code elimination and reduced allocation. In other words, inlining `foo` has a large long-term benefit, but a low short term benefit, whereas inlining `f` underneath the `foo` function call has a large short-term benefit. Incorporating a means for integrating the two forms of benefit into a single inlining heuristic is hard. A good formulation of the inlining problem should take into account both form of benefits, in addition to approximating the dependence between inlining decisions.

3.2.2 Engineering Challenges

In a completely hypothetical and theoretical compiler, there would exist a single inlining pass, that performs *only inlining*, sandwiched between other compilation passes. So effectively, in an "idealised" compiler, only that single compilation pass requires tinkering in studying the problem.

However, in real-world compilers, a "pure" inlining compilation pass simply does not exist. Instead, various other form of transformations are performed whilst inlining the code, especially dead code elimination (DCE) and function specialisation. If inlining a function results in less machine instructions than calling the function, the function ought to always be inlined (since instructions, on average, take roughly the same number of cycles when data is available readily in the L1 cache). Most functions, out of the box, will not meet this criterion. However, functions with large amounts of conditional statements and pattern matching can often have a large chunk of its code eliminated

```
(** The original function declaration **)
let rec map f l =
  match l with
  | [] -> []
  | hd :: tl -> f hd :: map f tl
;;

(** The original function call **)
map (fun x -> 2 * (x + 1)) [ 1; 2; ]

(* Step 1. make a specialised [map] function, [map'] (unused
           args are removed in future passes) *)
let map' _unused l =
  match l with
  | [] -> []
  | hd :: tl -> (2 * hd + 1) :: map _unused tl
in
map' () [ 1; 2; ]

(* Step 2. unroll once, then perform DCE *)
(2 * 1 + 1) :: map' _unused [ 2 ]

(* Step 3. unroll again, then perform DCE *)
(2 * 1 + 1) :: (2 * 2 + 1) ::  map' _unused [ 2 ]

(* Step 4. unroll again, then perform DCE.
   Loop unrolling reduced the function call into a simple constant. *)
(2 * 1 + 1) :: (2 * 2 + 1) :: []
```

Figure 3.5: Specialisation and inlining used together to indirectly perform loop unrolling via rewriting.

and meet this criterion. For that reason, it makes sense to have DCE run concurrently with inlining. Specialising right before inlining is also commonly done, to perform a combination of optimisation that is akin to loop unrolling. A combination of inlining and specialisation is often used to perform indirect loop unrolling in FP compilers, as illustrated in 3.5. By performing specialising and DCE alongside with inlining, the complexity of the exploration space is reduced substantially. Specifically, if a recursive function f contains B branches in a pattern match on its argument and U function unrolls, the number of inlining decisions that has to be considered with DCE is $O(U)$, as opposed to $O(B^U)$. Such design decisions are desirable from a pragmatic compiler construction standpoint, but it ends up cluttering attempts to study inlining as an independent problem.

On top of the purity aspect, when compiled with `-O3`, the Flambda compilation pass is executed over the IR several times, using the output of the previous round as its input. Each of the flambda round contains an "inline and simplify" pass that makes inlining decisions, so several obvious questions that arise: in which round should inlining be studied, and would this decision result in very different results? if each Flambda round can be thought as as $g_b \cdot f_{inline} \cdot g_a$ where g_b is passes before inlining and g_b is passes after inlining. Let N be the number of rounds, and flambda compilation pass can be modelled as, $(g_b \cdot f_{inline} \cdot g_a)^N$ for $N > 0$, where $f^2 = f.f$ and so on. Suppose there is a set of call sites $\mathcal{S}$ in a given program being optimised. The existence of compilation passes presents an interesting array of choices of study:

- **Studying inlining in the k-th flambda pass whilist assuming all passes as it is** - The number of call sites that require studying here is $O(|\mathcal{S}|)$. The other rounds are left as it is, so any attempts to modelling inlining decision is not only studying the behaviour of the ocaml runtime, but also modelling the behaviour of further Flambda passes and the choice of inlining decisions.

- **Studying inlining in all Flambda passes** - The inlining decision is call site parameterised,

```
; A pseudo-Flambda IR in lispy-syntax

; Assuming we have some function g1 defines as follows:
(define (g1 p) (g1' (+ p 1)))

; Step 1
(define (f a1 a2)
  (let (a (g1 a1)) ; <-- compiler is decides to inline the
                   ;     [g1] function call
    (let (b (g2 a2))
      ....)

; Step 2
(define (f a1 a2)
  (let (a (let [p a1] ; <-- compiler binds body's free variables to args
            ...       ;     to be generated
      )) ;
    (let (b (g2 a2))
      ....)))

; Step 3
(define (f a1 a2)
  (let (a (let [p a1]
            (g1' (+ p 1)) ; <-- compiler analyses the
                          ;     copied function body and decides.
      )) ;
    (let (b (g2 a2))
      ....)))
```

Figure 3.6: A rough depicition of FLambda's step when executing the inlining and simplifying compilation pass. Each arrow shows where the compiler is analyzing at each step.

in the sense that the choice of Flambda round is provided in addition to the raw inlining decision. On naive observation, the search space of this problem $O(N^{|\mathcal{S}|})$. On more careful inspection, it is evident that the search space is considerably smaller, since if s is a children of s', then $round(s) >= round(s')$, but nonetheless, at least a magnitude larger than $O(|\mathcal{S}|)$ due to the additional requirement to study the effects of the choice of inlining round.

- **Studying inlining in the k-th Flambda compilation pass and disabling inlining in all other rounds** - the search space in this option is similar to the first option. A reason for choosing this option is to investigate inlining independent of further composite decisions.

Choosing a specific form of inlining to study will most likely yield different results and knowledge. The eventual inlining policy might not be applicable across different rounds and compilation flags - the inlining policy is free to make assumptions about the subsequent compiler passes and the distribution of inputs to the inlining policy.

Another engineering challenge arises when attempting to make parallel inlining decisions throughout a function body. Flambda makes inlining decisions using a depth-first traversal, as illustrated in figure 3.2.2, whereby it recursively explore and inlines the body of a function before it stops. Modifying the compiler to perform parallel inlining decisions it not a straightforward.performing such inlining decisions in parallel is not straightforward.

To study inlining decisions in an offline manner (that is running sets of compilation batches first before, devising an inlining policy entirely offline, (Lopes 2010), rather than interleaving policy improvement simultaneously with compilation (Kulkarni et al. 2013)). A naive approach is to customise the compiler to dump all call site features and inlining decisions when compiling a program. However, there are two problems with the approach: (1) it assumes some form of independence between call sites and sacrifices any form of means of tying the relation between function call sites (2) it requires modification to the compiler (or modifications to dynamically

loaded runtime compiler library) whenever one decides to redesign features (3) it requires some means of unambiguously referencing inlining decisions and function call sites.

3.3 Problem Specification

Having looked at some performance indicator hypothesis and the landscape of the problem being studied, a concrete problem definition can now be formulated:

Can machine learning be used to learn a policy for making inlining decisions at the first (ie: 0-th) Flambda round for generating a (small) set of optimised native production builds, where the user can choose an optimal by the means of benchmarking, in the absence of binary-size and compile-time constraints?

- This inlining policy is free to define the set of flags passed to the compiler, as long as it does not changes the functional behaviour of the generated binary.[3]
- It is perfectly acceptable for compilation to be significantly slower, as long as it is acceptable within a production build's time budget.
- The performance of the inlining policy is evaluated based on the the *best performance* across the set of generated binaries.

In proposing a solution to the main problem defined by the project, the thesis attempts to answer several inter-related open-ended question in relation to function inlining:

- What data structure can represent the dependence between various inlining decisions?
- What metric can be used to quantify the benefits of inlining (or not inlining) a function?
- How can the relationship between a set of inlining decisions and the performance of a program be modelled?
- What is the nature of the inlining decision should the machine learning model(s) be trained to learn?

Some important non-goals of the thesis are:

- NOT TO produce an inlining pass that will work for arbitrary performance-related compilation options.
- NOT TO produce a fast binary in a reasonable compilation time suited for performance-tuning development cycles.
- NOT TO produce an inlining pass that will work for arbitrary Flambda rounds.
- NOT TO create a single general-purpose machine-learning guided inlining policy that can readily replace the current Flambda inlining heuristic.

[3] Flags like `-O3`, `-unroll-max-depth` is allowed, whereas flags like `-unsafe-string` are not.

3.4 Solution Overview

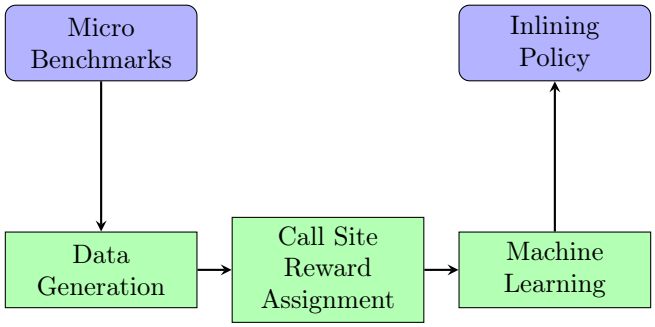


Figure 3.7: An overview of the training pipeline

A flow chart of the proposed optimisation pipeline is depicted in figure 3.7. While not shown in the flow chart, an implicit component of the pipeline is an inlining framework that facilitates communication between stages in the pipeline. The framework defines data structures to represent inlining decisions and a means of consistently label functions across various compilation runs.

Given a set of representation benchmark, the *data generation* stage iteratively compile each of the benchmark with different sets of inlining decisions. This yields, for each benchmark, a set of pairs, whereby each pair comprises of inlining decisions and execution statistics (a combination of execution time, garbage collection statistics and events sampled from `perf`).

In *call site reward assignment*, the execution statistics for a given benchmark is then fit to a model to yield the amount of "impact" of every inlining decisions. Every inlining decision will then be assigned a numerical value, denoting the impact of the single inlining decision alone towards the overall performance of the program[4].

The numerical value assigned to every function call site forms the basis for the training data for using *machine learning* to train function approximators. The modelled numerical values from the earlier stage are then transformed into labels for inlining decisions, that are in turn used to train a machine learning model for making inlining decisions.

[4]The observant reader might notice that this implies that nodes are possibly linearly independent, contradicting a claim made further up earlier. This will be cleared up in the relevant section.

Chapter 4

An Inlining Framework

An inlining framework is a methodology/mechanism that allows a compiler to request inlining decisions from an external source (by whatever means, such as dynamically linking a library and so on) and dump inlining decisions into a parsable format. Having a separate optimiser making inlining decisions reduces the overhead required in making changes (compiling a compiler is a lengthy process) and allows more iterative-based exploration of the search space. Such a framework also effectively allows inlining to be studied offline. As discussed in the background, there has not been a thorough study on developing an inlining framework. Support for the first functionality has previously been studied and used for research compilers. Jikes^{TM} Research Virtual Machine (Jikes RVM), used by Kulkarni et al. (2013) to study online function inlining decisions in the JVM.

This chapter describes a mathematical and experimental framework for specifying inlining decisions in a separate optimiser for general purpose high level programming languages' compilers. There will be theoretical discussion on generating *stable function call site labels* across manipulations such as inlining. The *inlining tree* data structure will also be presented as a data structure that encapsulates the relationship between inlining decisions and referenced throughout the report. Having presented the required theory, the implementation of the experimental framework is then presented.

4.1 Theory: Stable Path Labeling for ASTs

As described in the preceding chapter, one of the challenges in studying inlining is being able to generate stable labels for function applications in the program. As the Flambda IR is a tree-based IR, a generic methodology for labelling call sites in an AST can be used for this purpose. In this theoretical presentation, a mixture of mathematical symbols and haskell implementation for conciseness and relevant OCaml code will be used to explain certain quirks incurred in implementation.

Name generation for nodes in AST is commonly used in compilers and language processors. A common practice is to label nodes and generate variable names with ad-hoc unique identifiers, such as the `gensym` function as described by Augustsson et al. (1994).

```
int gensym() {
  static int i = 0;
  return i++;
}
```

However, when there exists multiple instances of AST / IR that have undergone a different set of manipulations, a stable labelling algorithm is important as it allows programs to identify and compare nodes that exist in both trees. In the context of iterative compilation, stable labelling enables comparison between of the sets of inlining decisions taken across different compilation runs.

Stability ensures that the label of a node is affected only by nodes that have been affected in earlier inlining operations. The `gensym` function does not satisfy the stability property, as demonstrated

by a simple example in figure 4.1.

```
                    /*****************************
                     * The original code snippet
                     *****************************/
                    int g() { ... }
                    int f(int x) {
                       return g(x + 1); // [1]
                    }
                    int main() {
                      int a =
                        g(2)     // [2]
                        + f(2);  // [3]
                      printf("%d\n", a); // [4]
                    }
/*****************************     /*****************************
 * After an inlining decision      * After a different inlining decision
 *****************************/     *****************************/
int g() { ... }                    int g() { ... }
int f(int x) {                     int f(int x) {
   return g(x + 1); // [1]            return g(x + 1); // [1]
}                                  }
int main() {                       int main() {
  int a =                            int a =
    g(2)     // [5] (inlined f(1))     f(1)     // [3]
    + f(2);  // [3]                    + g(3);  // [5] (inline f(2))
  printf("%d\n", a); // [4]          printf("%d\n", a); // [4]
}                                  }
```

Figure 4.1: The numbers in square brackets in comments refers to the call site identifier. The bottom two column shows the modified code after different inlining decisions are taken from the code snippet on at the top. `gensym` cannot stably label call sites across different set of inlining decisions.

A naive implementation that copies the identifiers from the transitive closure of u directly to v does not mantain the uniqueness property across mutations (and hence, incorrect), as shown in figure 4.2. In the given figure, `a`, `b` and `d` are function declarations and `e`, `h` and `i` are labels for function call sites. When `b` is inlined into call site `h`, the contents of `b` the label `a/d/e` ambiguously refers to two possible function call sites. It is entirely possible to inline the same function into two sibling declarations (For eg: a code snippet of the form $f(x) = g(x) + g(1 + x)$ where both instances of g are inlined). In LLVM and FLambda, variables names are rewritten to maintain with `gensym` to maintain the *Single Static Assignment (SSA)* property of the IRs.

The conflicts in labelling is a result of function inlining making arbitrarily copying different parts of the AST to function call sites throughout the AST. There is no inconsistency when initially constructing the AST however when loading it from source code, as long as the compiler is deterministic. Hence, ASTs that require labelling are constructed in two phases:

1. **Initialisation** Labels are not required for nodes at this stage. ASTs can be arbitrarily

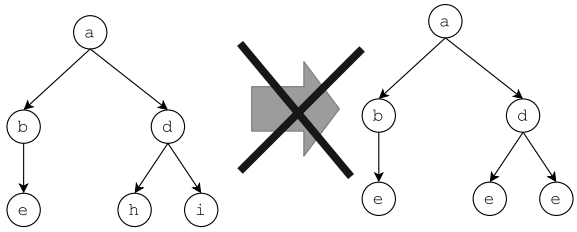


Figure 4.2: A naive id assignment for merge operation that does not work. Replacing nodes e and nodes f with the contents of node b results in a conflict in labels.

constructed, but all AST instances must be constructed similarly. All nodes are assigned a unique stamp using any methods, eg: using the the `gensym` function.

2. **Stable manipulation** At this stage, arbitrarily tree-manipulation is not permitted. A tree instance can be modified with a restricted set of operations. This ensures that identifiers do not get arbitrarily get created, maintaining the stability for reference by external applications.

Permitted stable manipulations are defined as follows:

- Tracked node addition $add(tree : T, u : V) \rightarrow T$ - New nodes can be added on the condition that their identifiers is a function of a element in the set of the its eventual siblings. It can be trivially shown that this enables an arbitrary amount of nodes to be added as children of a current function. For example, when performing function specialisation, function declaration being specialised can be copied as the sibling of an apply node, and the identifier of the function declaration shall be a labelled identifier with respect to the function node.
- Node Removal $remove(tree : T, u : V) \rightarrow T$ - Nodes can be removed without any restrictions. This corresponds to Dead Code Elimination (DCE) when simplifying the body of an inlined function.
- Node inlining - $inline(tree : T, u : V, v : V) \rightarrow T$ is defined as an operation that replaces node v with the transitive closure of node u, but excluding node u itself. For example, t_1 is obtained by running a merge operation on t_0, essentially replacing node g with the transitive closure of node b. In the context of IRs, this operation replaces call site v with the body of function declaration u

The initialisation phase correspond to parsing an OCaml source file, performing type checking, and converting it from lambda code to Flambda IR prior to any Flambda passes. The labels of nodes here are independent of the set of inlining decisions taken by Flambda. The stable manipulation phase corresponds to compilation and optimisation passes, specifically, the *inline and simplify Flambda pass*. At this stage, arbitrary sets of inlining decisions can be taken, and the path labelling algorithm should be able to consistently label function call sites across different sets of decisions. Concretely, an id function that returns the label of a node is required to fulfil the following logical properties:

Property 4.1.1. ***Mutation unrolling*** *where a is the merge destination of some source node d, $\exists(f, g).\forall(v, v') \in M(d, a).g(id(v')) = f(g(id(a)), g(id(v)))$. The identifier yields information about the total amount of merge that was required to arrive at a node.*

Property 4.1.2. ***Stability*** $\forall u \in V.\forall v \in C(u)$. *$id(v)$ is independent of $\{id(v') \mid v' \in C(u) \setminus \{v\}\}$ and its descendants. This ensures that the identifier of a node independent of unrelated merge operations.*

Property 4.1.3. ***Local uniqueness*** *After an arbitrary number of merge transformations, $\forall d \in V.\forall v \in C(d).\forall v' \in C(d) \setminus \{v\}.id(v) \neq id(v')$.*

It is trivial show that node removal will retain all of the above properties on a correctly labelled tree. Inlining is the main transformation that the algorithm requires special care studying for correctness. It is easy to show that the properties above are sufficient to define only a locally valid identifier. Locally identifiers are easier to track and allows room for more efficient algorithms. It is also possible to generate global identifiers from local identifiers, the specifics of this is discussed in a later subsection. A proof for tracked node addition was not formulated rigorously as it was not too critical for the correctness of the implementation, as compared to node inlining. Empirically, tracked node addition fulfils all the presented properties above[1].

4.1.1 Algorithm

Several building blocks for constructing the algorithm is shown in figure figure 4.3. $\mathcal{V}$ is a simple type definition for an AST that contains only function application and declaration, as the primary interest is in labelling function call sites. The *tree construction phase* builds the AST based on the IR from a previous compilation pass. Nodes are assigned unique identifiers $\mathcal{I}$ with an empty parents field, namely `gen_id () = (gensym(), [])`.

[1] in other words, I have not found an node addition example that defies the properties above

The main idea in the *stable manipulation phase* of the algorithm is to have the $\mathcal{I}$ data structure form a bottom-up representation of the set of merge operations that are formed. The algorithm comprises of two functions: (1) *LabelInline* constructing the identifiers due to inline operations and (2) *LabelUnroll* that outputs the identifier as a list of stamp. The two functions are shown in algorithms 1 and 2 respectively. The identifier representation in $\mathcal{I}$ is a compact variant that makes use of strong structural sharing. The *LabelUnroll* function simply turns it to a comprehensible form, and is used to prove the necessary properties above.

A alternative naive implementation of this idea is to simply have $\mathcal{I}$ represented as its unrolled form, namely $O(n)$ time for generating identifiers and $O(1)$[2] time for accessing the identifiers. The downside of this method is that after all transformations, the total memory required to store all identifiers is quadratic with respect to the number of transformations on the original tree. A proof can be trivially be considering the size of identifiers when inlining a function N times, ie: `["a", "a/a", "a/a/a", "a/a/a/a", ...]`.

An OCaml implementation, which takes into account of memory usage, is shown in 4.4. Both algorithms are conceptually very simple - the main challenge is proving their conformance with the above properties, whilst maintaining realistic memory footprints.

```
type id = (Int, [id])
type v  = Apply id | Decl Id [v]

let id = function
  | Apply x -> x
  | Decl (x, _) -> x

let children = function
  | Apply _ -> []
  | Decl (_, xs) = xs

let update_id node id' =
  match node with
  | Apply _ -> Apply id'
  | Decl (_, xs) -> Decl (id', xs)

let stamp ((x, _) : id) = x
let parents ((_, p) : id) = p
```

$$V = Apply\ \mathcal{I}\ +\ Decl\ (\mathcal{I}\ \times\ [V])$$
$$\mathcal{I} = (\mathbb{Z} \times [\mathcal{I}])$$

$$id : V \to \mathcal{I}$$

$$children : V \to [V]$$

$$update\_id : V \times \mathcal{I} \to V$$

$$stamp : \mathcal{I} \to \mathbb{Z}$$
$$parents : \mathcal{I} \to [\mathcal{I}]$$

Figure 4.3: Types and utility functions used to construct the path labelling algorithm and its proof, in both OCaml and logic syntax.

Input: Function declaration node, $d \in V$ and call site node $a \in V$
Output: $[V]$ a list of nodes with rewritten identifiers.
Function `LabelInline`(d, a)**:**
  $acc \leftarrow []$
  foreach *child* ***in*** $children(d)$ **do**
    $id_{child} \leftarrow id(child)$
    $parents' \leftarrow id(a) : parents(id_{child})$
    $id'_{child} \leftarrow (stamp(id_{child}),\ parents')$
    $acc \leftarrow concat(acc,\ [update\_id(child,\ id'_{child})])$
  end
  return acc

Algorithm 1: Creates a set of nodes with rewritten node identifiers upon merge.

[2]Realistically, any attempt to use the identifier will result in a linear-time operation. eg: equality checks. The only real exception is pointer comparison.

Input: Node label $\mathcal{I}$
Output: $[\mathbb{Z}]$ a list of stamps corresponding to the set of identifiers.
Function `LabelUnroll(`id`):`
  $acc \leftarrow [stamp(id)]$
  foreach *parent* ***in*** $reverse(parents(id))$ **do**
    $acc \leftarrow concat(Unroll(parent),\ acc)$
  end
  return acc

Algorithm 2: Recursively unrolls $\mathcal{I}$ from the representation to a list of stamps. This function is semantically equivalent to *concat(concat_map(Unroll, parents(id)), [stamp(id)])*

```
let inline d a =
  List.map (fun child ->
      let ps = parents (id child) in
      let ps' = id child :: ps in
      let id' = (stamp (id child), ps') in
      update_id child id')
    (children d)
```

```
let unroll id =
  let rec unroll' init id =
    List.fold_right
      (fun p acc -> unroll' acc id)
      (id stamp :: init)
      (parents id)
  in
  unroll' [] id
```

Figure 4.4: OCaml implementations of the labelling algorithm

Proposition 4.1.1. *The LabelInline function satisfies property 4.1.1*

Proof. Consider $(v, v') \in M(u, a)$ for some (u, a). Now let $f = concat$ and $g = Unroll$

$$
\begin{aligned}
& g(id(v')) = f(concat\_map(g,\ parents(v')),\ [stamp(v')]) \\
\leftrightarrow\ & g(id(v')) = f(g(id(a)),\ concat\_map(g,\ parents(id(v))),\ [stamp(id(v'))]) \\
\leftrightarrow\ & g(id(v')) = f(g(id(a)),\ concat\_map(g,\ parents(id(v))),\ [stamp(id(v))]) \\
\leftrightarrow\ & g(id(v')) = f(g(id(a)),\ g(id(v)))
\end{aligned}
$$

In the base case, in a freshly initialised AST, $parents(\phi) = \emptyset \leftrightarrow LabelUnroll(\phi) = [stamp(\phi)] = concat(concat\_map(LabelUnroll, parents(\phi)),\ [stamp(\phi)])$.

□

Corollary 4.1.1. *LabelUnroll will never return an empty sequence.*
Proposition 4.1.2. *The LabelUnroll function will always terminate for all well-defined identifiers.*

Proof. $LabelUnroll(x)$ will not terminate if and only if the the $parents(x)$ has a cyclic connection back to x. Hence, it suffice to prove that there is not cycles in the parents relation. This can be proven by induction. Assume there are no cycles in the directed graph defined by edges $\bigcup_{u \in V} \{\ (u, p) \mid p \in parents(u)\ \}$, that is the edges define a Directed Acyclic Graph (DAG). Identifiers generated as a consequence of merge operations adds new nodes to the existing identifier's graph, maintaining the graph's acyclic-ness. Since the base case is an identifier with no outgoing edges, hence a DAG, the proposition is inductively proven. □

Proposition 4.1.3. *LabelInline satisfies property 4.1.2.*

Proof. In the proof of proposition 4.1.2, it has been shown that the forms a DAG. On detailed concrete observation, it is trivial to notice that the DAG formed by nodes curated via *LabelInline* points strictly towards parents. Hence, the $\forall\phi.LabelUnroll(\phi)$ is dependent only on the itself and parent identifiers and hence, independent of its sibling nodes and descendants. □

Proposition 4.1.4. *LabelInline satisfies property 4.1.3.*

Proof. First, defining identifier equality as $\forall x \in \mathcal{I}.\forall y \in \mathcal{I}.x = y \leftrightarrow LabelUnroll(x) = LabelUnroll(y)$, with the later equality following conventional list equality semantics. The proposition can be proven inductively.

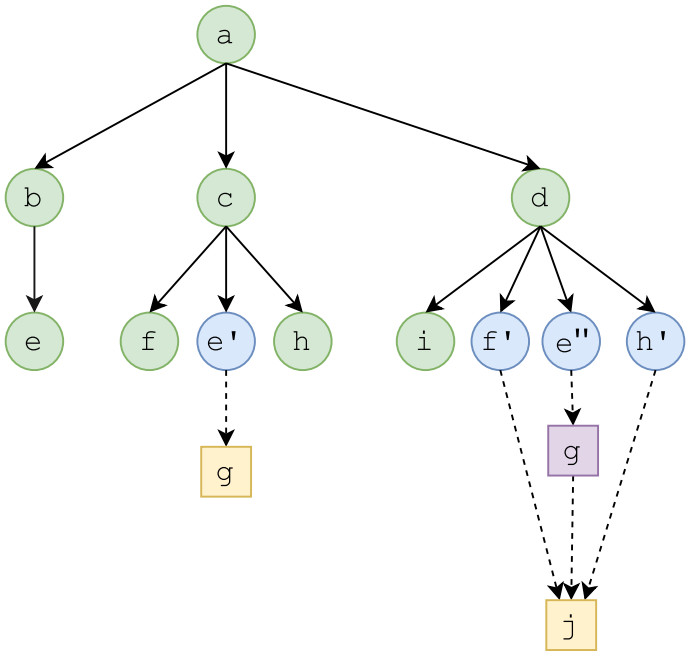


Figure 4.5: A possible AST after several merge operations. Solid edges are edges in the AST, whereas dotted edges are edges in the identifiers $\mathcal{I}$

Consider two identifiers $id(a_x)$ and $id(a_y)$ with common prefixes. This denotes that they originated from a common sequence of merge operation. The inductive assumption states that $prefix(id(a_x)) = prefix(id(a_y)) \implies suffix(id(a_x)) \neq suffix(id(a_y))$, where for some suffix and prefix functions such that $LabelUnroll(id(a)) = concat(prefix(id(a)),\ suffix(id(a)))$.

Consider the case where $|Unroll(id(a_x))| = |Unroll(id(a_y))|$. By inlining some function declaration u into call site a_x, some $v_x^{(i)}$ is unveiled. Since $|Unroll(id(v_x^{(i)}))| < |Unroll(id(a_y))|$, therefore $id(a'_y) \neq id(v_x^{(i)})$. Now by merging u into a_y, $v_y^{(i)}$ is obtained where $\forall i.|v_x^{(i)}| = |v_y^{(i)}|$

For $|Unroll(id(v_x^{(i)}))| = |Unroll(id(v_y^{(i)}))|$, it is necessary for $suffix(id(a_x)) = suffix(id(a_y))$. The requirement contradicts with the initial assumption, and hence, the assumed argument is unsatisfiable. By contradiction, it is shown that if the uniqueness constraint hold $\forall a \in V$ where the length of identifiers for its children are equal, then it also holds in the general case. □

As all the propositions and corollary presented above were shown inductively, a very useful result follows: It is possible to run this merge for an arbitrary number of times whilist retaining the properties in propositions 4.1.1, 4.1.2, 4.1.3 and 4.1.4. In other words, this inlining path labelling algorithm can be used simultaneously in multiple inlining pass whilist retaining all of the above properties.
It is trivial to show that creating a single new identifier with *LabelInline* uses $O(1)$ memory, based on algorithm 1. Consequently, the memory requirement for storing the entire tree is $O(N_{Inline} \times E_v[|C(v)|])$. $\mathcal{I}$ recursively defines a bottom-up representation of inlining operations conducted on the AST. Walking from a node up to the root of a mutation tree unveils the source of merge operations that has been undergone to unveil a certain node in the tree. This idea is illustrated in the an example in figure 4.5. In the given figure, simply walking along the dotted edges of `e''`, it can be deduced that `e''` in unveiled from inlining call site `g` into call site `j`, and inlining that `g` all into the body of function declaration `d`. Structural sharing (not illustrated in the diagram) in this example is attained whereby both instances of $id(g)$ occupy the same section in memory.

A naive equality implementation for equality check is to $eq(x, y) = eq(Unroll(x),\ Unroll(y))$. This uses linear-memory and would most likely suffice for general scenarios. However, when the size of the $\mathcal{I}$ tree is big (due to very aggressively inlining), a more memory efficient variant of an equality check can be performed by simultaneously walking both tree from the two given identifiers.

4.1.2 Implementation in the OCaml Compiler

Property 4.1.3 defined above only guarantees the identifier uniqueness locally between siblings. To obtain global uniqueness, the global identifier node can be defined a pair of the path to the node and the node's identifier, namely $\mathcal{I}_{global} = [\mathcal{I}] \times \mathcal{I}$. For example, the unique global identifier of node e'' in figure 4.5 given by $id_{global}(e'') = ([id(a),\ id(d)],\ id(e''))$, which is formed by a combination of the path from the tree root to node e'' and $id_{node}(e'')$.

The primary purpose of developing a path labelling algorithm is to be able to (1) pass custom inlining decisions to the compiler and (2) read the set of taken inlining decisions by the compiler at different Flambda rounds. As it has proven that this algorithm fulfills all the properties after any number of flambda rounds, this algorithm works for whichever round in the Flambda pass. For completeness, the type interface of the local and global identifiers are as follows. The actual implementation differs slightly in type of several fields, that use more specialised types rather than `int` or `string` for type safety.

```
type compilation_unit =
  { id           : int;
    linkage_name : string;
  }

type stamp =
  | Plain_apply of int
  | Over_application of int

type local = {
    compilation_unit : compilation_unit;
    stamp            : stamp;
    parents          : local list;
  }

type trace_item =
  | Decl of local
  | Apply of local

type global_id = trace_item list
type action = Inline | Apply
type decision = (global_id * action)
```

Using the a set of inlining decisions each of which described by the `decision` type above, an override decision set can be constructed. At a function call site, the compiler will attempt to look for a decision in the decision set, with a time complexity of $O(N \times E[|trace|])$ (A performance optimisation is possible, and is described in a later section). This decision set can also be used to report the set of inlining decisions that are taken by the compiler unambiguously, useful for offline inlining analysis.

4.2 Theory: Inlining Tree

The inlining tree is data structure that can be used to encode inlining decisions taken when compiling a program. The inlining tree allows for encoding information about high-order functions, namely having function declarations beneath inlined functions and declarations, alongside with function applications and inlined functions. The data structure has the following recursive type definition:

```
type node =
  | Declaration of declaration
  | Apply_inlined_function of inlined_function
  | Apply_non_inlined_function of non_inlined_function
and declaration =
```

```
  { declared  : Function_metadata.t;
    children  : node list;
  }
and non_inlined_function =
  { applied   : Function_metadata.t;
    apply_id  : Apply_id.t;  (* refers to [id] in path labelling *)
  }
and inlined_function =
  { applied   : Function_metadata.t;
    offset    : Apply_id.t;  (* refers to [id] in path labelling *)
    children  : node list;
  }

type root = node list
```

The main distinction between a typical AST / IR and this is that it preserves the hierarchical nature of inlining decisions in a program (eg: inlining a function application can unveil other function call sites that can be inlined as well), and shows the amount of inlining that was required for a function to unveil the set of applications and declarations that it has. In an AST / IR, upon inlining a function call, the original function application is discarded and removed from the AST / IR. The inlining tree takes into account the hierarchical / nested nature of inlining decisions, hence preserving the dependence between inlining decisions.

Function applications in this inlining tree are leaf nodes. Function declaration and inlined applications can be either leaves or regular nodes in the tree. While this might sound counterintuitive, in Flambda (and lambda code, in fact), some functions are primitive operations under the hood. Some examples are arithmetic operators (`+`, `-`) and polymorphic comparison operators, and hence, are not candidates for function inlining and not included in the inlining tree. It is also possible that functions return constants, got DCE-ed, or simply had the transitive closure of its function applications inlined exhaustively.

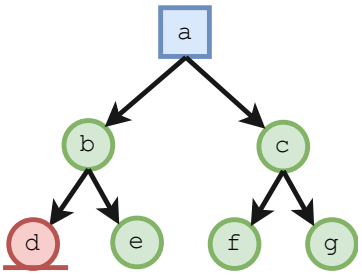
A subtle detail about the inlining tree is that the set of declaration ids and *apply_id* of apply nodes in the descendants of the function declaration is not necessarily the same the children of an inlined function call applying the said function. This is because code can be DCE-ed, but another possible factor is that function declaration can add arbitrary function declarations into the inlining tree due to function specialisation. For this reason, while no guarantee can be said about the set of declaration ids, the set of apply ids in the children of an inlined node, in a bug-free implementation, should be a subset of the set of apply ids of apply nodes obtained from the transitive closure of its function declaration[3]. An example of inlining tree an its corresponding implementation is shown in figure 4.2.

To facilitate the discussion in subsequent notions, the inlining trace will be used. The inlining trace is a path along the inlining tree, encapsulating the set of decisions taken from the root node of the inlining tree up to a given decision site (or function declaration). An informal syntax expressing inlining traces (to facilitate presentation and discussion) will be used, comprising of `<a>`, `{a}` and `<a|>` corresponding to a inlined node, function declaration and an apply node in the inlining tree respectively. Eg: `<a> -> <b> -> <c|>` represents a trace where (1) the call site `a` is inlined; (2) the call site `b` beneath `a` is inlined; and (3) the call site `c` beneath `b` is *not inlined*.

4.2.1 Constructing From Inlining Decisions

Following the modification to the compiler discussed in an earlier section, the compiler reports the set of taken inlining decisions at the end of compilation. The inlining tree can be constructed incrementally using raw inlining decisions, similar to adding "phrases" into a trie. The algorithm assumes that all elements in the set of inlining decisions are consistent with one another. An example of two inconsistent decisions are `<a> -> <b|>` and `<a> -> <b> -> <c|>`.

[3]It was mentioned bug-free implementation, as this is not true in practice. When inlining recursive functions with free variables, Flambda in ocaml 4.06.0+flambda "incorrectly" inlines the original variant of the function that has not been simplified.




```
let d x = (* ... a lot of code ... *)

let e y =
 2 * y
;;

let b x y =
  d x / e y
;;

let f x = 100 * x
let g y = 300 * x

let c x y =
  f x + g y
;;

(* before inlining *)
let a () =
 b 1 2 - c 3 4
;;

(* after inlining *)
let a () =
 (d 1 + (2 * 2)) - ((100 * 3) / (300 * 3))
;;
```

Figure 4.6: The inlining tree at the left corresponds to the function declaration rooted at a after inlining some function calls in its body.

Whilst the time complexity of this construction algorithm is $O(N \times |trace|_{average})$, the space used by the inlining tree reflex the "true" amount of inlining complexity present in the program. A program that has a large inlining tree probably has a lot of inlining decisions to make, relative to counterpart with a smaller tree. For example, the branching factor of the inlining tree represents the average number of calls a function declaration possesses prior to any compiler transformations, and the depth of the inlining tree denotes the amount of nested inlining that the compiler has searched. Such information is not attainable from inspecting raw inlining decisions themselves, or looking at the AST.

4.2.2 Generating Compiler Overrides

The mapping of inlining decisions to inlining tree is a many-to-one mapping, implying the existence of redundancy. For example, consider the inlining tree that will be constructed from the following two sets of decisions: (1) `{(<a> -> <b> -> <c>)}` and (2) `{(<a> -> <b> -> <c>) , (<a> -> <b>), (<a>)}`. The first decision set contains the minimal amount of decisions, and cannot be reduced further (ie: no redundancies). The two inlining traces in the second decision set is redundant, but the set represents *all* the inlining decisions that was taken in the program. This gives rise to two forms of decision sets:

Definition 4.2.1 (Minimal decision set)**.** The smallest possible set of decisions to represent a set of inlining decisions. Every inlining tree have a unique minimal decision set.

Definition 4.2.2 (Maximum decision set)**.** The largest possible set of decisions to represent a set of inlining decisions.Every inlining tree have a unique maximum decision set

The first decision set is a minimal decision set, whereas the second decision set is a maximal decision set. Sets with different level of redundancies exists as well.

Correctness Measures

An inlining tree is said to be *sound* if the set of decisions taken when compiling the program, should it exist in the inlining tree, should be consistent. For the inlining overrides mechanism to be useful, it is necessary for the inlining tree-guided overrides to be sound. This is an important test that has to be performed to verify the inlining framework's correctness.

An inlining tree is said to be *complete* if all the inlining decisions taken in the program can be queried in the inlining tree (but does not necessarily have the right answers). The completeness measure is a means of evaluating the coverage of inlining trees. Completeness is a sometimes desirable, but incompleteness in inlining tree is important for allowing exploration.

Generating incomplete decision is used for exploring the space of inlining trees. Flipping an *apply* node into an *inline* node as a new inlining override will almost certainly result in an incomplete inlining tree, because it lacks information about the inline flipped node's children. By using this as a set as an override decision set, the resultant taken decision set will contain new information about the set of nested inlining decisions that can be taken, useful for search space exploration.

A scenario where soundness and completeness is desirable (and required) is when the taken decision set is passed into the compiler as a override decision set. A user would do this when to recover certain compilation artifacts, such as the assembly file or IR dumps.

Algorithm

To generate the maximal decision set, the procedure simply has to recursively walk the inlining tree, adding the inlining trace constructed from a node into the decision set. A subtlety that requires care is to pass inlining trace downwards as it recursively descends the child nodes. Generating the minimal set of slightly tricker - adding an element to a decision set should be delayed to the point where there exist a non-apply or inlined node when descending the tree. The algorithm to convert an inlining tree to the maximal decision set is shown in A.2.

4.2.3 Usage

Tree Diff

One useful property for the inlining tree is that it is possible to diff-ing between tree can potentially yield more useful information about the set of inlining decisions that were taken. This is especially useful when comparing two set of inlining decisions where one leads to significant speed improvement over the other. Tree-diff groups related difference in the same "diff", effectively reducing the number of differences that matter. Conceptually, there are 3 general forms of diffs in inlining trees:

- Nested diff. This form of diff is obtained when one tree possess an leaf apply node, whilist the other tree contains a non-leaf inline node corresponds to that exact function. Whilist there are further decision items beneath the leaf node, they carry much less significance compared to the lowest common ancestor with a flipped decision.
- Simple diff. This form can easily be obtained from comparing raw inlining decisions, and is usually not the most interesting form of diff. The difference is obtained by literally flipping a leaf inline / apply node.
- Missing diff. This form of diff occurs when the there exist nodes in the two trees which are exclusively available only in one of the trees (ie: not nodes that references the same apply id). This form of is not easily observable from raw inlining decisions.

Each diff contains three components - the common ancestral trace, left-only components and right-only components. The algorithm to compute the diffs in the tree corresponds to walking the two trees (known as left and right) simultaneously. The nodes are split into 3 categories, left-only, right-only and same. nodes in left-only and right-only are yield as diff results and the algorithm recursively descents into the nodes under the "same" category. The key idea of the algorithm is to selectively recursively walk down both trees simultaneously. The children of the roots nodes are

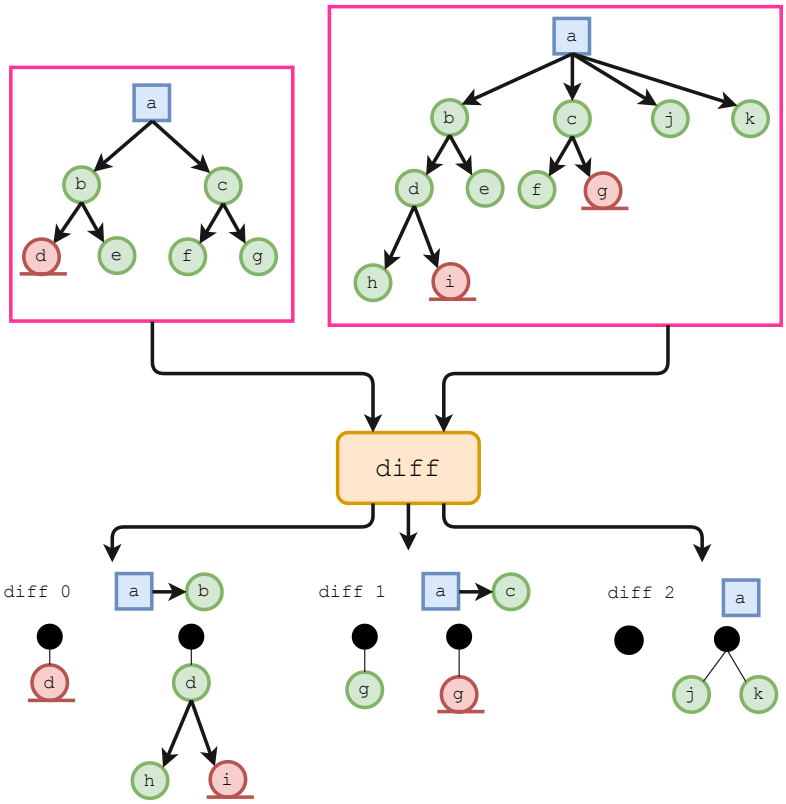


Figure 4.7: An example of diff-ing two inlining trees. This leads

diff-ed using a shallow-equality measure, and nodes that are exhibit shallow equality are recursively descended to look for more diffs. A slight subtlety that the algorithm has to handle is to correctly label the common ancestors between the two pieces of diff when recursively descending inlining trees. An OCaml implementation illustrating this algorithm can be found in appendix A.1.

An visualised example of the results of executing diff is illustrated in figure 4.7. Diff 0, 1 and 2 correspond to the nested diff, simple diff and missing diff respectively. An obvious use of this diff is to compare inlining decisions, but more notably, when two compilation produces significant speed differences, this mechanism for comparing inlining decisions can quickly narrow down the set of inlining decisions that lead to the massive speed improvement.

Searching

Consider a compiler that has been given an inlining overrides by the user – at a function call site, what is the complexity of querying the inlining overrides for a relevant inlining decision, in addition to any preprocessing complexity?

The preprocessing step's complexity can vary. If a compiler represents the decisions as an exhaustive decision set, it will require no preprocessing, but the runtime complexity is larger. Querying the complete decision set for a single inlining decision requires a brute force search, using a simple equality comparator. The worst-case complexity is $O(N_{exhaustive} \times E[|trace|])$. On the other hand, when given a minimal set, the comparison operator, instead of an equality check, is a prefix-equality check. This comparison is $O(N_{minimal} \times E[|trace|])$ and since $N_{minimal} \leq N_{exhaustive}$, it is always asymptotically better to reduce a complete decision set to a minimal set prior to compilation.

The observant reader would have found that most of the comparison is repeated work. Instead of searching queries, a simple preprocessing step will be to construct the inlining tree from the any semantically equivalent decision set. The inlining tree can be treated as a trie (De La Briandais 1959), and inlining queries is akin a phrase. The complexity of looking a query is then reduced to $O(E[|trace|] \times E_v[|\mathcal{C}(v)|])$, where B is the branching factor of the tree.

4.3 Experimental Infrastructure

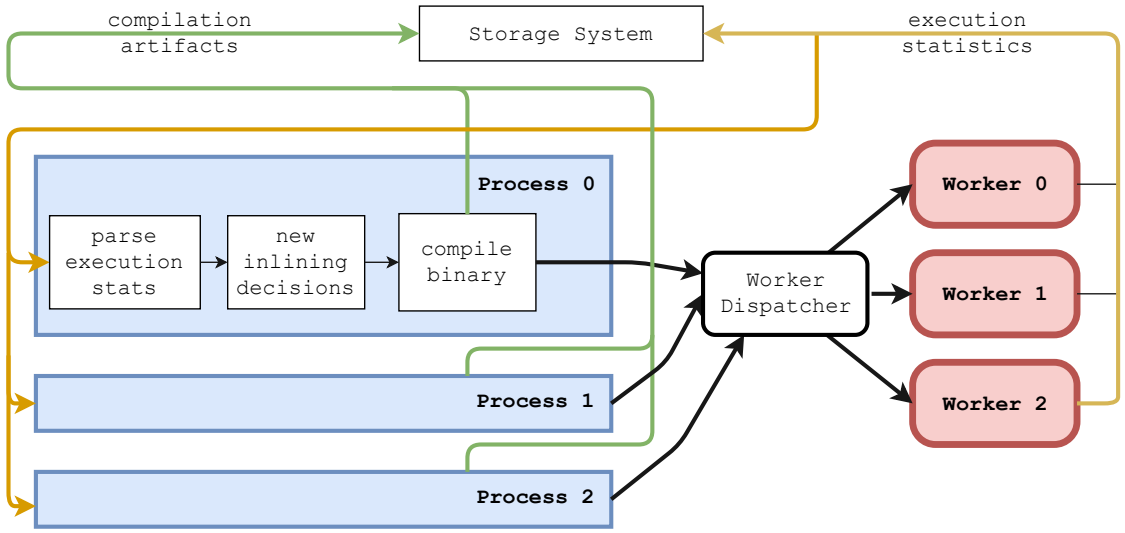


Figure 4.8: Architecture for evaluating inlining decisions

The experimental infrastructure manages the communication and worker dispatch between a set of independent optimising processes and a set of workers, in addition to systematically collecting execution statistics and compilation artifacts produced in the optimisation processes. This experimental framework makes no assumptions about the process' underlying algorithm for inlining decisions' search space exploration. An overview of the setup the overall infrastructure is shown is depicted in figure 4.8. There is no unified means of communications between components - the worker dispatcher communicates with workers using ssh over an ethernet in a local network, whereas the storage system is abstractly represented using a set of library functions.

To ensure that each worker has at most a single program, a worker dispatcher sits between the processes and workers. It is possible to have the worker dispatcher be a process running in the local network, which is preferred where the different optimisation processes are running on different physical machines. However, when all optimisation processes reside in the same physical machine, it suffices for the worker dispatcher to be an agreement to use a lockfile. A directory in the master machine houses several files that serve as locks to ensure mutual exclusion in acessing the workers. A lock file can be created by creating the lock with with the flags `O_CREAT`, `O_EXCL` and `O_RDWR`, when the lock is not attained, the file creation will fail. The lock is released by simply removing the created the file.

The infrastructure is implemented with a mix of shell scripts and OCaml, using the async (LLC & Contributors 2018*a*) library to exploit cooperative threading and type-safety for asynchronous programs. The detailed specifications of the workers, compiler versions and so on are listed in appendix D.

4.3.1 Workers

Three homogeneous workers are used in this project for running program benchmarks. Each of this worker has a intel i7 Haswell Quad-core processor (SMT is enabled, hence 8 virtual cores), 500GB HDD and 8 GB of DDR3 RAM. Note that whilst workers need not be a single machine in batch execution terminology, in the context of this project, a worker refers to strictly a single machine, which is permitted to run only a single benchmark a single point in time (regardless of the number of cores it has). Benchmarks in this project are limited to statically linked programs that has minimal runtime dynlink dependencies.

To ensure the cleanliness of the data and instruction caches, the benchmarks are run only on the first core of the machines. To prevent other applications in the user space from utilising the cache, the kernel is booted with flags `isolcpus=0`, which disallows user-space threads from being

scheduled on core 0. (Inc 2018) As of writing, preventing the kernel from scheduling processes in the kernel space on the excluded core is not possible. Benchmarks are executed with commands in the form `taskset 0x1 ./main.exe arg1 arg2` to explicitly instruct the kernel to execute the mentioned userspace process in the first core.

When issued a job, the worker executes the mentioned benchmark $N + 3$ times, where N is a user-defined configuration. The first N runs, it is important to choose $N > 1$, as the first run is most likely to be slower than the 2nd run due to an unwarned-up I-cache. By ensuring that other user-space programs do not get scheduled to the specific core, it is very unlikely that a user-space process does not get scheduled to the physical core. For iterative compilation and exploration experiments, this N is set to 2 to allow more thorough exploration of the search space, when measuring for benchmarks, N is set to a 5 to be able to detect potentially bimodal distribution in execution times. The remaining 3 rounds are executed with (1) `perf` whilst monitoring a select-set of counters such that counters get at least 50% coverage (2) `perf` whilst monitoring a larger set of counters and (3) `perf` whilst monitoring the call graph and then dumping the garbage collection statistics. It matters less which run the GC stats are collected, in practice, as long as they are not done in one of the N time-critical runs. The GC stats and sizing of the OCaml heap is a deterministic process with respect to the allocations performed in the program.

4.3.2 Storage System

The general approach taken in the framework (and the project on a whole) in data collection is simple: measure as much as possible - storage is cheap, data is expensive. The data collected per execution includes: compiled executable, S-expression representation of inlining decisions, a human-readable dump of the flambda IR after execution, perf call graph and perf counters. Rather than having the process decide choose a directory to dump the collected data, the processes store the data in a central repository. This central repository, in the even of a cluster setup, can utilise a Network File-System (NFS) setup, and when using just a single machine, can just be a single directory. Every optimisation process is initially assigned a `rundir` and for every compilation and execution. When compiling a binary and executing a benchmark, the process passes in a `work_unit_id` to the framework, with the below type signature. The framework automatically saves the compilation artifacts and execution stats in `<central-repository>/opt_data/<step>/<substep>/{compilation_artifacts.tar, perfdata.tar, execution_stats.sexp}`. With a consistent means of data storage, the data can easily be referenced whenever required.

Note that the data obtained is compressed. For a single execution, the size of binary is often around 2-3 MB, inlining decisions for large programs can go up to 1MB and the largest of all, is the call graph obtained from perf, which can go up to 95MB for a 10-second run (almost independent of the size of the program). Without any compression, execution of a single benchmark would require 105MB – A 500GB disk would run out of space after merely 5000 executions. Assuming there are 3 workers running in parallel, and each benchmark takes 60 seconds in total (due to multiple runs), the disk space would be exhausted after $5000.0 * 60.0/3 = 100,00$ seconds, which is equivalent to roughly 1.157 days. This is not desirable! Hence, the framework compiles data files collected on the fly. The decision to compile compilation artifacts across substeps and steps independently is motivated by the ease of implementation and access, despite not providing the best compression ratio[4]. Suppose the artifacts throughout a process is compiled together, during future analysis, reading a single execution will incur the overhead of decompressing all steps' compilation artifacts. Compilation reduces the size of the artifacts to roughly 2 MB in total, which lasts roughly 57.9 days. Swapping out harddisks on every two months is much more desirable and reasonable as opposed to daily.

[4]Compressing more things simultaneously, in general, results in smaller file sizes on average

4.4 Evaluation

4.4.1 Path Labelling Algorithm

Functionally, the algorithm works as expected, based on hand-picked test-cases. This is expected as a formal proof of correctness has been presented. To measure the asymptotic behaviour of the algorithm, a benchmark that recursively merging the leaf child of a node by itself in an AST with 100 nodes and a branching factor of 3[5] is evaluated. The benchmark is written in C++ using a minimal tree data structure. This AST mutation is similar to those when inlining a recursive function repeatedly in the declaration, akin a loop unrolling operation. Two variants of this benchmark is written - one that uses raw pointers and one that uses smart pointers. In addition to that, a naive implementation of the algorithm, whereby identifiers $\mathcal{I}$ are represented directly using a list of identifiers $[\mathbb{Z}]$.

The memory usage is measured using valgrind Nethercote & Seward (2003) and the execution time is using the bash time utility. The relative execution time and memory usage in tables 4.1 and 4.2 is obtained by dividing by the case where there is only a single inline operation. As shown in the table, the execution time and memory usage of the algorithm with structural sharing (in both variants) increases linearly with respect to the number of inline operations, as opposed to the naive implementation.

Inline Ops	Raw Pointers	Shared Pointers	Naive Impl.
1	1.000	1.000	1.000
10	1.011	1.019	1.016
100	1.111	1.203	1.715
1000	2.074	2.966	63.492
10000	12.759	22.532	6164.270
50000	57.100	103.724	153866.345

Table 4.1: Relative in memory usage of labelling algorithm

Inline Ops	Raw Pointers	Shared Pointers	Naive Impl.
1	1	1	1
10	1	1	1
100	1	1	1
1000	1	1	2
10000	2	3	73
50000	6	12	1469

Table 4.2: Relative execution time labelling algorithm

The proposed algorithm is generalisable to arbitrary programming languages that supports function inlining to label function call sites, as it does not rely on any properties of OCaml. A draft publication of the algorithm, which is presented as an AST-labelling algorithm, is attached at the end of the thesis.

4.4.2 Infrastructure

The framework typically works as expected. Due to compression of collected data, a 500GB typically gets used up after 2 months, similar to the predicted 69 days as above. Throughout the course of iterative compilation experiments, there was not any cases where the program simply deadlocks for inexplicable reasons. There was several rare occurrences network connection outage (due to an occasionally faulty network switch) - this results in a deadlock when the master machine deadlocks while copying data or retrieving results from / to the workers. Bugs in the OCaml compiler sometimes results in the compiler simply crashing and even worse, executes infinitely and consuming a lot of memory, resulting in a thrashed operating system. Handling compiler crashes is easy, as a failed "make" command. Network failures and operating system thrashing are handled with the `timeout` command in bash. The command automatically terminates all processes in its group after if it does not exit within a given time limit. This is especially useful for indirect compilation that doesn't invoke `ocamlopt`, but rather, build tools like `make` or `jbuilder`. Fixing the bugs in the compiler is beyond the scope of the project. In an iterative compilation scenario, the process will simply have to assume that inlining decisions that result in a compilation failure (timeout or compiler crashing).

[5]Source code for this benchmark is available at https://github.com/fyquah95/stable-tree-labeling

4.5 Summary

A lot of ground was covered in this chapter regarding the build-up work required to study function inlining, namely:

1. a stable path-labelling algorithm to reference function call sites across different set of inlining decisions in compiling a program.
2. an implementation to override and read inlining decisions in the OCaml compiler.
3. the *inlining tree* data structure to represent inlining decisions taken during compilation. The inlining tree retains the hierarchical relationship between inlining decisions.
4. an experimental setup for running and collecting data from experiments and benchmarks.

Chapter 5

Generating Data on Inlining Decisions

The first step of the optimisation pipeline is to generate data on inlining decisions. Methods to systematically exploring the space of inlining trees of a given set of microbenchmarks will be presented in this chapter, by building on top of the inlining framework presented in the previous chapter. The chapter conclude with an evaluation of the program performance from iterative compilation and some statistics of the collected data.

NB: The term "state" will be used to refer to inlining tree, only in this chapter. In optimisation and exploration algorithms, "state" is commonly used to refer the current solution being evaluated.

5.1 Perturbing Inlining Trees

Perturbations are transformations that turns a given inlining tree to a "neighbouring" inlining tree. An "ideal" perturbation should be able to transform an inlining decision back and forth where required, concretely, all perturbations $f \in P$ should have an its inverse $f^{-1} \in P$.

Designing such a perturbation function for finitely-bounded and regular data structures (should as a fixed-length vector or a singly-linked list) is easy, but it is hard, in the case of irregular data structures like trees, which is the case for inlining trees. An example of a simple non-ideal perturbation is a function that flips a non-leaf *Inline* node in an inlining tree to an *Apply* node. There is no guarantee that when that transformation is reverted (by flipping the node again), the set of inlining decisions underneath that node will remain the same, as they might have previously been manipulated.

A slightly weaker requirement for the set of transformation to be controllable, that is it is possible to manoeuvre to all states from any given state using P. In other words, if s' is a neighbour of s, there definitely exist a chain of successive *neighbour* call on s' which can result in a set that contains on s. This weaker requirement is formally defined as:

$$s' \in neighbour(s; P) \implies \exists\ (k >= 1)\ .\ s \in neighbour'(k, s';\ P)$$

where

$$neighbour'(k, s; P) = \begin{cases} \{f(s) \mid f \in P\} & k = 1 \\ \bigcup_{s' \in neighbour(s;P)} neighbour'(k-1, s';\ P) & otherwise \end{cases}$$

The two main building blocks perturbations for the inlining tree are hence is flipping a node flipping a leaf node, f_{flip} and backtracking the parent of a leaf node, $f_{backtrack}$. The algorithm for performing these transformation is left out due to its simplicity. f_{flip} turns a leaf apply node to an inline node, whereas $f_{backtrack}$ turns the parent of a leaf node from an *Inline* node and *Apply* node. An illustration of the two perturbations is shown in figure 5.1. It is trivial to show that by using a combination of perturbation functions composed of he two given building blocks, it is possible to explore the space of all inlining trees for a given program, hence, P is a controllable set of perturbations. That is, is if the iterative compilation strategy makes a bad exploration step that results in a performance dip, there exists a series of transformation that would allow it to eventually recover to a good state.

```
let b p q = x p - y q

let c p q = x (y p) + p

let f p q =
 let p' = a (p + q * 2) in
 let q'= b p' p in
 c p' q'
;;

ignore (f ())
```

Figure 5.1: Three possible inlining trees. Blue nodes corresponds to a declaration, green ones corresponds to inlined function and red one being non-inlined functions. Figure 5.2 corresponds to a the original inlining tree. Figure 5.3 shows the case where several leaves are flip. It is entirely possible (and likely), whereas figure 5.4 shows the result when `b` is backtracked.

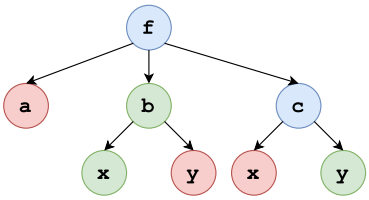


Figure 5.2: Original tree

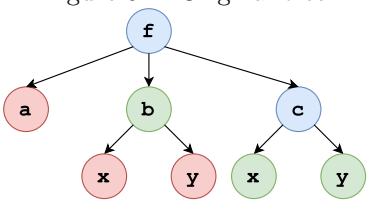


Figure 5.3: Tree with flipped nodes

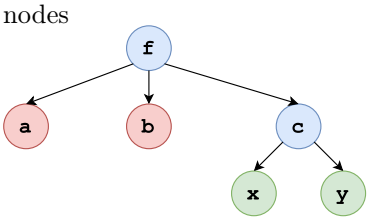


Figure 5.4: Tree with backtracked nodes

The implementation of this two perturbation functions are fairly straightforward are fairly straightforward tree manipulation algorithms. To uniformly sample the set of perturbations from a given state[1] (or inlining tree), the perturbation can be selected by randomly choosing a number from 1 to $|leaves(s)| + |\{parent(s') \mid s' \in leaves(s)\}|$. The main subtlety that requires attention here is to omit declaration nodes from the set of nodes, as f_{flip} and $f_{backtrack}$ do not operate on declaration nodes.

5.2 Iterative Compilation

Iterative compilation (Knijnenburg et al. 2002) is a compilation method that generate many versions of a program and decide upon the best choices by actually executing them and measuring their execution time. The original idea was presented in the context of modifying loop unrolling factors. Iterative compilation is used here to compile the same program while iteratively modifying the set of inlining decisions over different compilations. Iterative compilation requires some form of strategy to systematically search the state space (in this case, the set of inlining trees).

[1] not to be confused with uniformly sampling the set of states

5.2.1 Simulated Annealing

Simulated annealing (Aarts & Korst 1988) is a form of probabilistic method to minimise a given target function, also known as the energy. The method operates by exploration via perturbating over states and eventually settling on a local minima. At every iteration, the probability of accepting (hence, transitioning) the new state is define in terms of the energy difference between the two states and the temperature (defined in terms of the time step). The amount of exploration is decreased over time using a temperature schedule, gradually shrinking the search space across iterations. The probability of transitioning from one state to another is defined as follows:

$$P(n, s, s') = \begin{cases} 1 & energy(s') < energy(s) \\ \exp(-\frac{\text{energy(s') - energy(s)}}{temp(n)}) & otherwise \end{cases}$$

Unlike many optimisation methods, simulated annealing does not require one to calculate the partial derivative of the target function with respect to the optimisation parameters.

To adapt simulated annealing for the context of spatial exploration the context of exploring the space of inlining decisions, the a uniform sample over the set of perturbations P is used as the neighbour function. In principle, the neighbours function is required to form an ideal perturbation set. However, it was empirically found that perturbation functions satisfying only the weaker argument generates decent program execution times.

The temperature schedule is chosen as shown in equation 5.3, with the parameters[2] in figure 5.5. Every version of compiled code is executed 5 times and the speedup over baseline (that is when the override decision set is empty) is used as the energy of simulated annealing.

$$temp(n) = exp(-iter \times \frac{log(Temp_{max}/Temp_{min})}{N_{steps}}) \tag{5.3}$$

$T_{max} = 0.04$	the maximum simulated annealing temperature 0.04
$T_{min} = 0.0005$	the minimum simulated annealing temperature
$N_{steps} = 300$	the number of iterations in simulated annealing exploration

Figure 5.5: The parameters when iteratively compiling with simulated annealing.

The main downside of simulated annealing is that it is inherently sequential, that is, the state and result of perturbation of (n) is dependent on state *and* results in step $(n-1)$. This is potentially hamper-ring, where there is a lot of workers that is available to work in parallel. A method to mitigate this problem is to run M independent parallel explorations, where M is the number of workers. The downside of this method is that the generated data is highly-biased towards to the initial base, as opposed to running a single long exploration process with $M \times N$ iterations. Iteratively compile a program with simulated annealing requires $O(N \times E[Time_{exec} + Time_{compile}]) = O(N)$ parallel time.

5.2.2 Random Walk

The space of inlining trees can also be explored data by performing a uniform random walk over the space. Over a set number of iterations, the set of neighbours of the current inlining tree, without regard for the results. This approach samples the state space more uniformly than those of simulated annealing, which biases towards the local minima (and inlining decisions).

Unlike the simulated annealing, the state at iteration n is independent on the result of the state at iteration $n-1$. Hence, all N inlining trees / states can be generated ahead of time without waiting on the results of any benchmarks. As a result, iteratively compile a program with random walk requires $O(max(N \times Time_{compile},\ E[Time_{exec}]))$ parallel time.

[2]These parameters have not been thoroughly tuned, as it is not the primary goal of the project.

Benchmark	Initial	Simulated Annealing	Random Walk
floats-in-functor	9.320	**4.238 (54.524%)**	6.354 (31.824%)
lens	9.798	**4.470 (54.376%)**	4.471 (54.371%)
kb	8.351	7.711 (7.663%)	**7.696 (7.837%)**
sequence	12.707	12.040 (5.253%)	**11.856 (6.697%)**
hamming	12.627	**11.985 (5.080%)**	12.000 (4.965%)
lexifi	15.275	**14.612 (4.337%)**	14.653 (4.069%)
bdd	13.065	**12.635 (3.295%)**	12.722 (2.626%)
almabench	10.536	**10.283 (2.397%)**	10.474 (0.585%)
quicksort	13.920	13.726 (1.397%)	**13.723 (1.415%)**
fft	8.014	**7.920 (1.172%)**	7.941 (0.905%)
sequence-cps	14.286	**14.272 (0.098%)**	14.277 (0.063%)
kahan-sum	8.617	**8.616 (0.011%)**	8.622 (-0.058%)

Table 5.1: Summary of results from iterative compilation with simulated annealing and random walk.

5.3 Results

The exploration on inlining decisions was performed *only* on the entry-point module of the function [3]. This was probably a poor design decision made early on in the project, as it reduced the available search space of several experiments. Nonetheless, exploring only in the entry-point module still yields non-trivial performance improvements.

As a consequence of running different runs of iterative compilation on different programs is that a set of pairs of inlining trees and execution statistics[4] is generated for every program. This data will be used in later chapters for modelling and machine learning.

A table of the results in the execution times is shown in table 5.1. Figures 5.6 and 5.7 visualises the execution time over the set of explored states with simulated annealing and random walk respectively. The figures demonstrate the distribution of execution times obtained from the compilation strategies.

The `floats-in-functor` and `lens` benchmark demonstrates that the best observed performance is most more than 50% better than the default performance, suggesting that the default heuristic can be improved significantly in call sites that exhibits certain specific structures. These speedups conforms with the suspicions presented in hypothesis 3.1.1 and 3.1.2.

The attained speedup in `float-in-functor` is due to inlining a loop implementation of calculating the modulo of a floating point value, reduce in a significant reduction in the number of minor / major collections. A single inlining decision in the critical loop reduced the number of major and minor collections from 26291 and 848 to 13061 and 235 respectively (9.6s to 6.3s). Massive improvements in `lens` can be attributed to inlining a seemingly boring-looking function, that returns a value block with 2 integers. The speedup is due to compounded effects from other optimisation pass, which leveraged the knowledge about the two fields. While it is interesting to inspect benchmarks with large speedups, performing fine-grain analysis of individual speedup is however usually very tedious and difficult, especially for benchmarks with a small amount of speedup. An improvement >4% can be observed half the examples, 1 - 3 % is observed in a third of the example. Two of the examples demonstrate little-to-none improvements.

Readers familiar with compiler constructions will find the results in `sequence-cps` rather surprising – wouldn't inlining do a very good job in simplifying CPS code? The 2 possibilities are: FLambda is already doing a pretty decent job in inlining CPS-code, so regardless whatever decision was taken in the first pass, the subsequent pass does a really good job in inlining or (2) the k-continuations are not exposed for inlining. Based on eye-balling the inlining tree, the results suggest the former – the final inlining trees look very similar after the final FLambda pass, and the

[3] Entry-point module in OCaml does not really mean anything, but the microbenchmark are written with a definite entry point

[4] combination of execution time, speedup over baseline, garbage collection data and perf event counters

`sequence` benchmark manages to achieve a non-trivial performance gain. The main imprvoement in the sequence structure of CPS code is more likely in the realm of stream-fusion (Coutts et al. 2007, Mainland et al. 2013). It is some what surprising to find `kahan-sum`, a benchmark filled with floating-point operations, to have hardly any speedup. This is because of the choice to deal with top-level modules exclusively. Most of the floating point operations in `kahan-sum` is not in its top level modules.

The nature of the data obtained from simulated annealing and random walk also differs, in which simulated annealing data tends to reject exploring bad examples altogether. This effect is most evident in `bdd`, where simulated annealing (figure 5.6) barely explored performance in the regions of 14.00s (the best peak is approximately at 12.75s), compared to random walk (figure 5.7). The presence of "weaker" examples is useful for running machine learning at a later stage. In the `sequence` benchmark, random walk managed to obtain a better peak performance than simulated annealing, suggesting the possibility that simulated annealing *can* get stuck at a local optima. This claim, however, is neither empirically nor properly proven.

In most cases, the distribution of execution times visually forms a sum of independent (but not identical) gamma distributions[5]. An interesting for the probability density function for execution times would be:

$$f_{pdf}(t) = \frac{1}{Z} \sum_{i}^{N_{peaks}} \gamma(t - peak_i; k_i, \theta_i) \tag{5.4}$$

where Z is a normalisizing constant such that $\int f_{pdf}(t)dt = 1$. N_{peak} as the number of call sites where the inlining decisions result in a significant change in the execution time. By eyeballing figure 5.6, this pattern can be observed in `almabench` ($N = 1$), `bdd` ($N = 2$), `hamming` ($N = 5$ or 6) and so on. The line of thought suggests some interesting ideas, eg: in `bdd`, the inliner managed to capture one "significant inlining decision" and completely missed another. The noise is produced by a combination of kernel processes[6], I-Cache status and quite possibly, inlining decisions that do not matter.

5.4 Summary

The space of inlining trees for a given program can be explored by using a combination of perturbations via f_{flip} and $f_{backtrack}$. Iterative compilation of microbenchmarks can be achieved by perturbing the space of inlining trees for the given program by systematical sampling using simulated annealing ($O(N)$ parallel time) and random walk (with $O(max(N \times Time_{compile},\ E[Time_{exec}])$) parallel time). It is found that iterative compilation can improve the performance of generated code quite significantly, suggesting that improvements can be made to the inlining policy employed by the compiler. Iterative compilation produces a set of data comprising sets of pairs of inlining trees and execution statistics is generated, ready to be used in the next step of the optimisation pipeline.

[5] Not co be confused with the sum of iid gamma distributed random variables

[6] recall that it is not possible (as of time of writing) to prevent kernel space processes from being scheduled into a processor

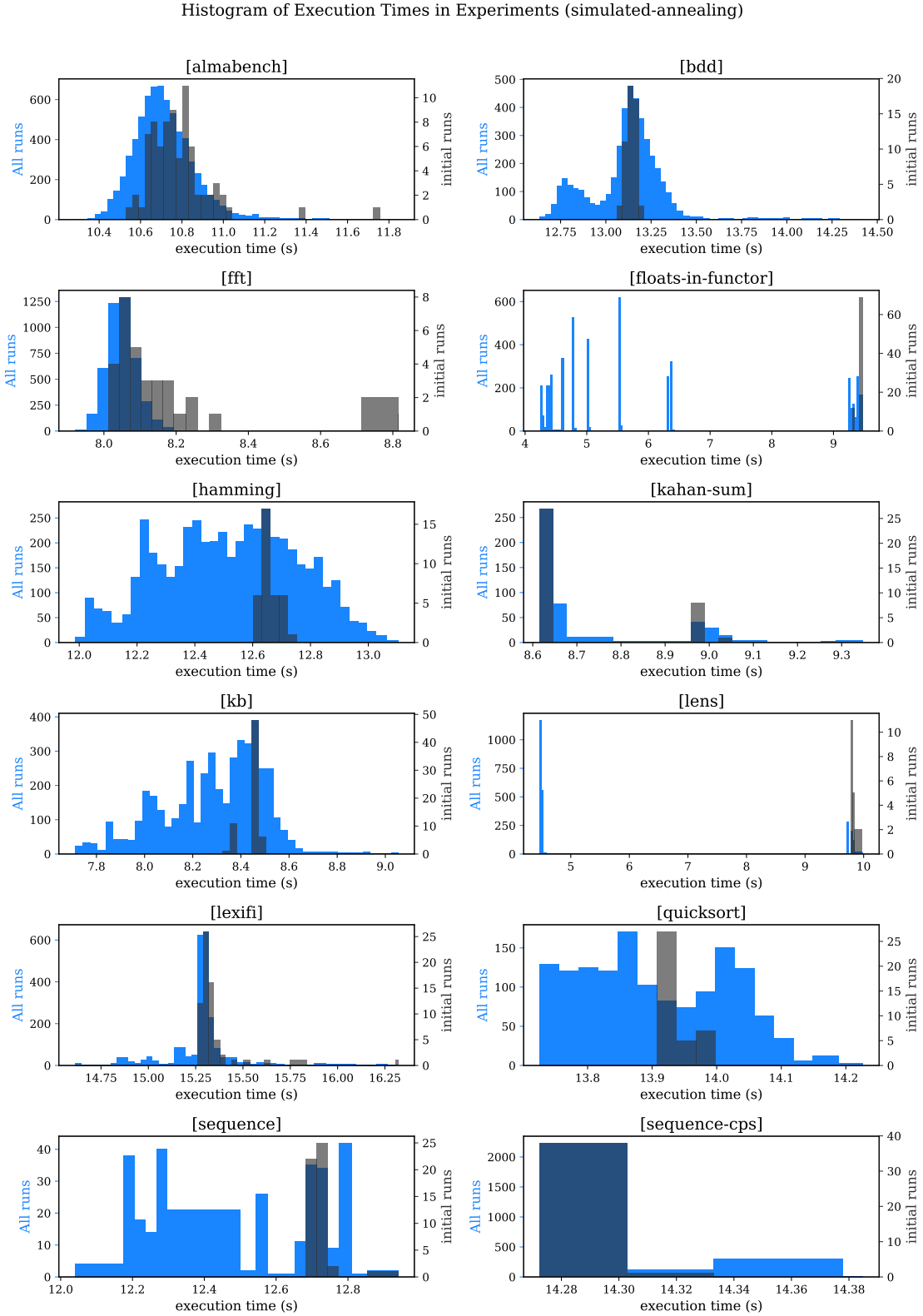


Figure 5.6: Execution time over benchmarks using exploration with simulated annealing. The bin size is approximately 0.03 seconds

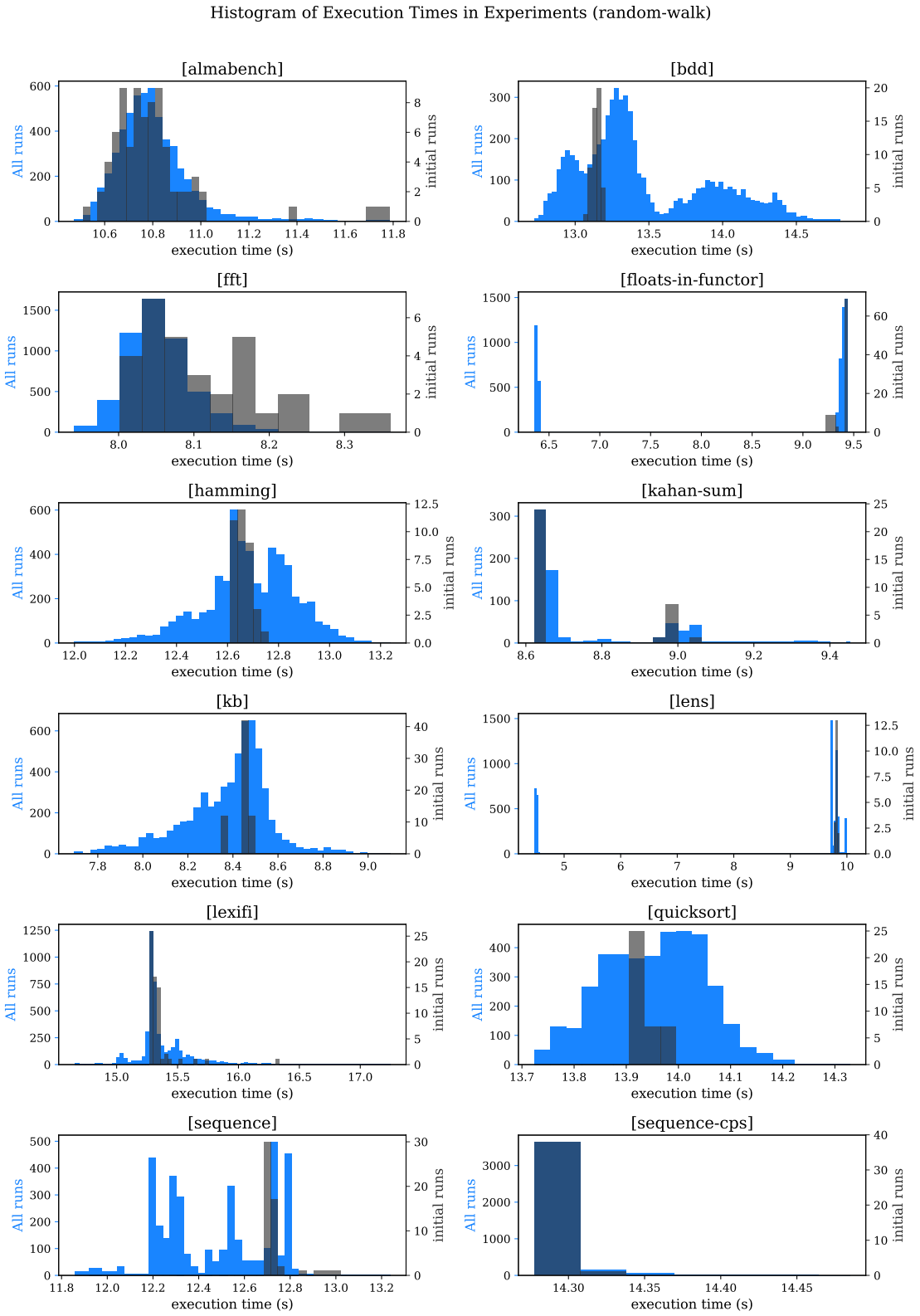


Figure 5.7: Execution time over benchmarks using exploration with random walk. The bin size is approximately 0.03 seconds

Chapter 6

Assigning Numerical Values to Inlining Decisions

This chapter will discuss how sets of inlining tree and execution time pairs can be modelled into individual inlining decisions and their associated value. As discussed in the background, the amount of time spent in functions (typically in profile guided optimisation (PGO)) and the size of the function body (almost always done in static inlining heuristics) is used to construct inlining heuristics. A fundamentally different approach is taken here - that is to forgo all prior assumptions about program behaviours, and build a empirical predictive model to quantify the attained reward from inlining decisions.

6.1 Incremental Tree Space Exploration

The inlining decisions problem can be viewed as an incremental tree space exploration problem. The goal of tree-space exploration is to find the tree with the highest value, using whatever means of navigating the tree space. In incremental tree-space exploration, the exploration originates from the root, and the tree is incrementally constructed.

Namely, the exploration problem begins with a single root node $u \in \mathcal{V}$. Now, an action $a \in \mathcal{A}$ can be operated on the node u, denoting a choice made in the decision process. Carrying out this operation will yield a local reward $r_{u,a} = \mathcal{R}(u, a)$ and a set of children nodes $\mathcal{C}(u, a) = v_0, v_1, v_2, ...$ The exploration will then proceed recursively in the set of children nodes. By taking inspiration of Markov Decision Problem formulation in reinforcement learning (Sutton & Barto 1998), this structural exploration problem can be formulated as follows:

$\mathcal{A}$	set of actions.
$\mathcal{V}$	set of all nodes
$\gamma \in [0, 1]$[1]	discount factor
$\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$	Local reward from taking a certain action from a given node.
$\mathcal{C} : \mathcal{S} \times \mathcal{A} \to \wp(\mathcal{S})$	Set of nodes unveiled from an action at a given node.

γ defines how much further decisions impact decision made in a certain node. The goal of the incremental tree exploration is to learn a policy that finds a tree that maximises the total-value of the tree. Concretely, the goal of this dynamic tree exploration problem is to find a policy $\pi : \mathcal{S} : \mathcal{A}$ such that maximises the total value of the tree. Where $\mathcal{C}_\pi(s) = \mathcal{C}(s, \pi(s))$, the optimal tree value is:

$$V_{\pi*}(s) = \mathcal{R}(s, \pi(s)) + \gamma \sum_{s' \in \mathcal{C}_{\pi*}(s)} V_{\pi*}(s') \tag{6.1}$$

There is no constraint on $\mathcal{C}$, except that all nodes in the tree is unique (which is a characteristic of trees, anyways). Note that it is possible (though not necessary) that there exist sequence of actions that causes the tree expansion to proceed indefinitely. Using dynamic programming to build a navigation policy bottom-up has a time-complexity of $O(E_v[|\mathcal{C}(v)|]^{depth} \times |\mathcal{A}|)$ for an arbitrary tree space. Consequently, navigating the tree space requires some form of heuristic policy π to choose an action such that (1) $|V_{\pi*}(root) - V_{\pi}(root)| < \epsilon$ and (2) $\forall v \in \mathcal{V} \,.\, \pi(v)$ runs reasonably fast.

This set of equations defined above is not to be confused with tree-search algorithms, such as alpha-beta pruning (Knuth & Moore 1975) or Monte-Carlo Tree Search (MCTS) (Kocsis & Szepesvári 2006) and reinforcement learning. The goal of those algorithm is to learn a policy that yields the highest *long-term reward*, whereas in the problem defined above, the interest is in a tree-structure that yields highest *total reward*, based on the formulation given above. Those fields an algorithms are interested in the *path* and the search space is described by a tree. In our formulation, we are interested in a *tree* where the search space is described by a set of trees.

6.2 Expanded Tree

The inlining tree data structure is useful for visualisation and analysis, as presented in chapter 4. However, the inlining tree still suffers from the distant-node inconsistency problem described in chapter 3, whereby inlining decisions in the body of a function declaration affects (1) the attained benefit from the function call and (2) the set of further inlining decisions beneath inlining the said function call. In the context of the above tree-space exploration, it is not possible to derive a pure function $\mathcal{C} : \mathcal{V} \times \mathcal{A} \to \wp(V)$ for an inlining tree due to such inconsistency! This problem is illustrated in an example shown in figure 6.1. In the example shown by the figure, the only "real" difference in inlining decision made is that node call site g beneath declaration of c is not inlined in the one tree, but not inlined in the other. However, due to only 1 inlining decision, the two inlining trees look drastically different. The definition of $\mathcal{C}(d, Inline)$ becomes ambigious, as it will yield different sets of nodes for both trees, despite compiling the same program.

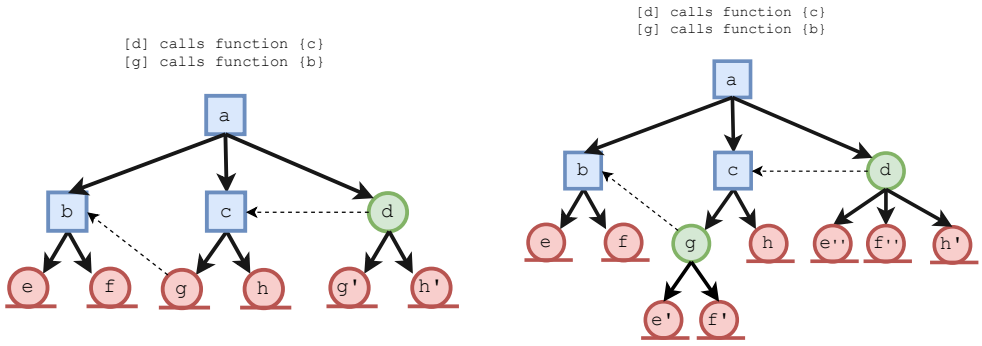


Figure 6.1: Example illustrating distant-node inconsistency in inlining trees. Call site g beneath declaration c is inlined in the left inlining tree, but not the one at the right.

The solution of this problem is to "expand" the inlining tree and recursively unroll the sequence of inlining decisions that was executed in the declaration site to unveil a function application underneath an inlined function call and the set of inlining decisions that were taken in the original function declaration. The result of expanding an inlining tree is the expanded tree. An example demonstrating the distinction between inlining and expanded trees is shown in figures 6.2 and 6.3. In the given example, the grey e is a replicated inline node from the original function declaration b and the gray d node is recover as part of the absolute sequence of inlining decisions to unveil call sites f and g.

Both types of trees have very similar type definitions, hence, most library code for tree manipulation can be shared. In order to utilise the type system to statically differentiate expanded trees and inlining trees, we have used phantom types (Cheney & Hinze 2003) in the tree's declaration to indicates whether it is an inlining tree or expanded tree.

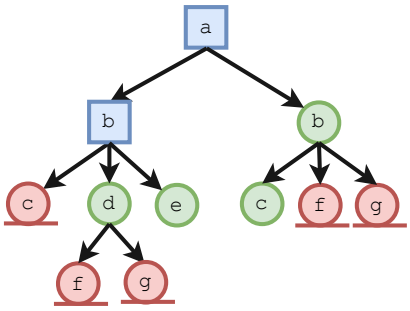


Figure 6.2: Inlining tree for a program.

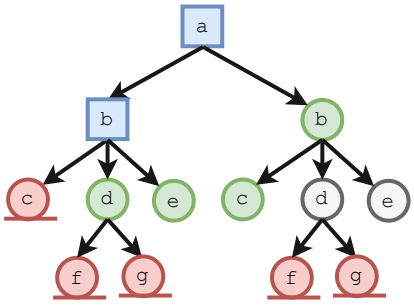


Figure 6.3: Its corresponding expanded tree Nodes in grey are replicated nodes.

6.2.1 Transforming from Inlining Trees

In an idealised scenario, the declaration for all functions are available in the source code. In which case, the rough idea of an algorithm is to "stack" the subtree of the declaration on top of the set of children, with the leaf apply nodes overlaying, as shown in figure 6.4. In practice, there can be several problems with this algorithm: (1) the declaration of all applied functions are not necessarily accessible. This is possible due to functions that are not included in the inlining tree's exploration space, such as functions from external packages or the OCaml standard library; (2) the inlined version of the function is not the same as the declared one (possible due to bugs in the compiler). Detecting (1) is easy. (2) is a necessary condition for the `apply_id` equality between the leaf apply nodes of the declaration and the children nodes of the *Inline* node is a necessary condition - so the equality can be used as a fuzzy check to detect (2).

To compensate for the unavailable function declarations, best-effort basis expansion can still be done by using expanded the identifier of inline and apply nodes. Due to property 4.1.1 of the `apply_id`, some of these nodes can still be partially retained. The rough idea of this expansion is to compare the labels of parent and child nodes, and add nodes that should be in between them. If a parent node of has an `apply_id` of `<a>/<b>/<c>`, the child node must have an `apply_id` of the form `<a>/<b>/<c>/...` with a non-empty suffix following their common prefix (due to proof 4.1.1). Elements in the non empty suffix can be converted into nodes in the expanded tree. An illustration of this idea is shown in figure 6.5. This best-effort expansion will fail to capture leaf inline nodes in the original declaration. For example, in figure 6.3, the grey *Inline e* apply node will not be replicated in the expanded tree, but the d node will still be present, if this form of best-effort expansion is used.

The resulting conversion algorithm uses a combination of the declaration replication and best-effort expansion. While the algorithm on its own is not trivial, its details are not important for the understanding of the thesis. The implementation of the algorithm is attached in appendix A.3. The worst-case space complexity of the algorithm is $O(d \times n)$. While this worst-case is an asymptotic concern, it does not happen frequently in any of the experiments to pose as any practical concern.

Another algorithm that is worth mentioning is turning an expanded tree back into a set of inlining decisions or an inlining tree. The details of the algorithm is not important, but it is worth nothing that there is no one-to-one mapping to inlining trees (hence, no one-to-one mapping to inlining decisions). It is possible to generate an overrepresented set of inlining decisions from an expanded tree, similar to how decision sets can be generated from an inlining tree. Care has to be taken in the comparison check, but it is mainly tedious programming work, and hence, left out of discussion here.

6.2.2 Properties and Limitations

A key assumption in transforming an inlining tree to an expanded tree is that the impact of inlining is a commutative operation (where equality is defined by performance equality). In other words, the sequence in which call sites are inlined is carried out to unveil a certain function call does not matter, as long as the sequences, when concatenated together, are similar. The model breaks

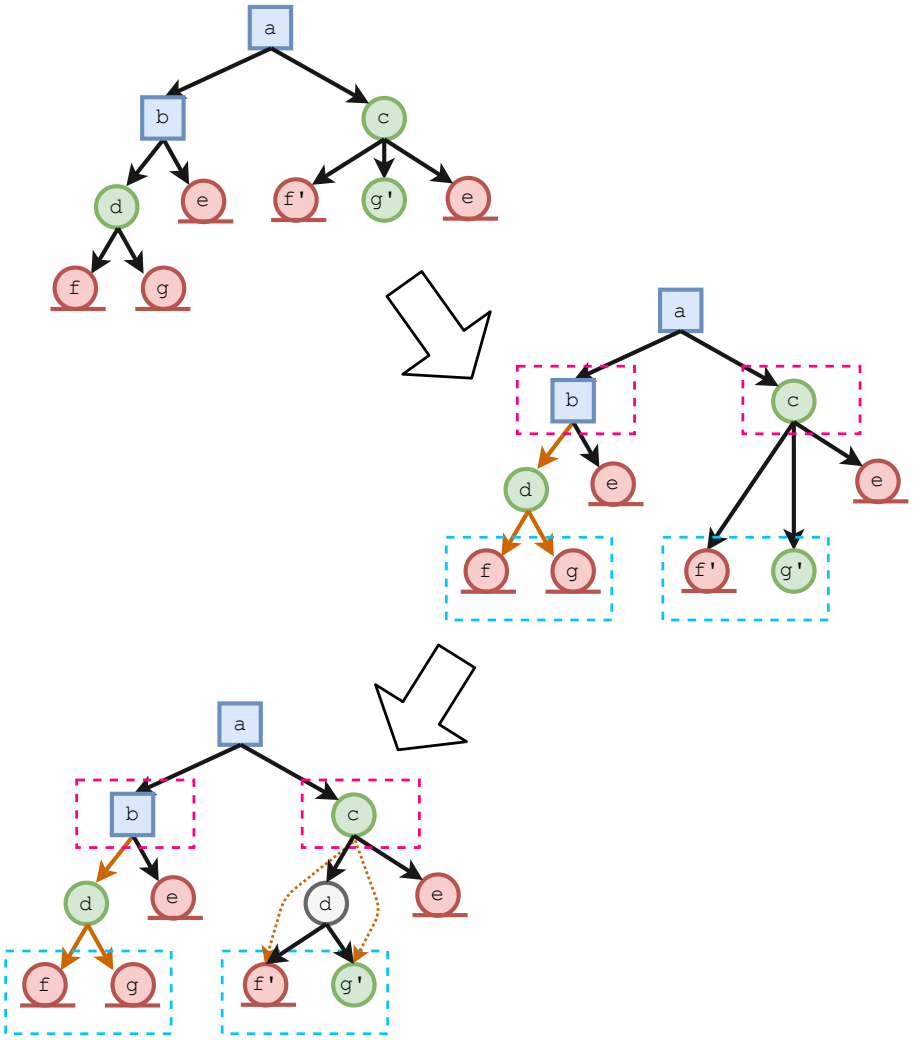


Figure 6.4: A visualisation of the steps taken to transform a replicate the nodes from the declaration to function call site.

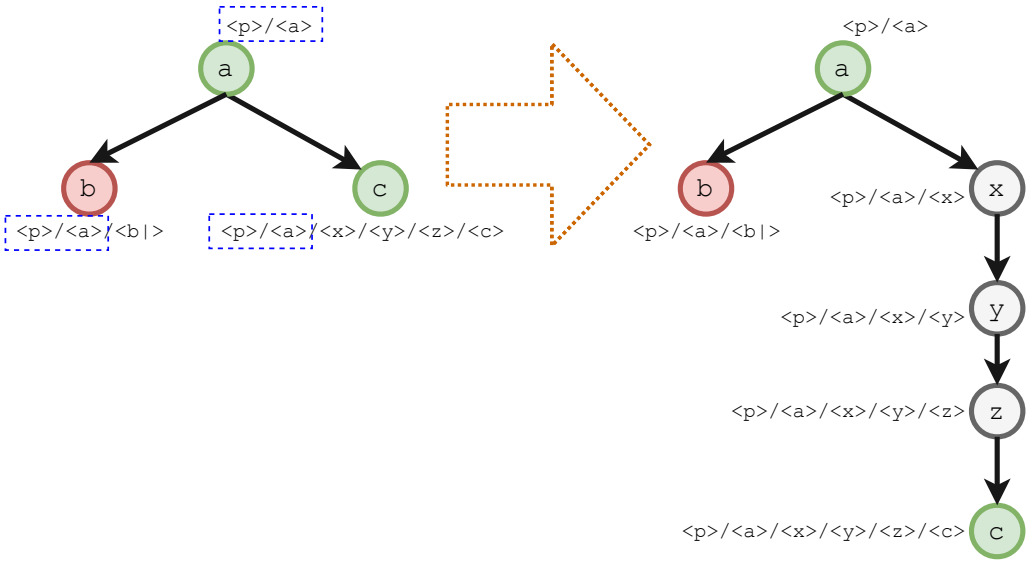


Figure 6.5: Best effort basis of path expansion when the declarations cannot be found.

down if this assumption is not true, but based on our knowledge about how DCE and how type information is used in compilation passes, this should not pose as a problem in practice.

A corollary of replicating inlining decisions in the declaration is that the depth of nodes relative to its lowest ancestral declaration node no longer define the amount of inlining that the compiler has performed when traversing into a function call. The depth of the nodes now represents the absolute number of function inlining, including those executed indirectly in the function declaration, that was executed to unveil a function call. This results in a mostly consistent representation - whereby the children nodes of a node in the expanded tree is independent of the structure of other nodes (or decisions taken in incremental exploration).

The representation is only mostly consistent, as it is still possible more nodes to be missing due to (1) compiler bugs; or more convincingly (2) nodes that are removed via DCE. Inconsistency due to (2) is possible when function inlining is performed in different orders. A weaker guarantee can be given however, that is it is possible to define a finitely-sized "upper-bound set" where $\forall v \,.\, \mathcal{C}^*(v) \in \mathcal{C}(v)$. For the purpose of the project, we will assume that in-existence of this limitation and define $\mathcal{C}^*(v) = \mathcal{C}(v)$ for modelling.

6.2.3 Formulating as an Incremental Exploration Problem

Following the discussion on the properties and limitations of the expanded tree, it should be evident that exploration across the space of expanded tree for a given program can be in fact modelled as an incremental tree-structure exploration problem with the following definitions:

$$\mathcal{A} = \{Inline, Apply\} \tag{6.2}$$

$$\forall v \in \mathcal{V}.\mathcal{C}(v, Apply) = \emptyset \tag{6.3}$$

It is worth noting that this problem is a special-case in the tree-structure exploration described above, because there exists an action effectively terminates exploration over the subtree of a node. This translate to the compiler's choice to stop inlining altogether. This is very useful as an "off-switch" in preventing inlining from going on indefinitely, a useful fallback when inlining recursive functions or when importing excessively large code blocks.

6.3 Problem Definition

As discussed in the chapter's introduction, the primary objective is to assign numerical values to inlining decisions. Having expressed the problem in terms of exploration of the tree space, the predictive modelling problem is now reduced as follows:

For a given benchmark and γ, learn $\mathcal{R}(v, a)$ such that $\mathcal{V}_{\pi*}(v, Inline) > \mathcal{R}(v, Apply)$ if it is better to inline the call site v, and the converse otherwise. M instances of expanded trees and their corresponding execution statistics are given as demonstrations.

The size of the benchmarks used for training are finite and small ($|\mathcal{V}| \approx 20,000$), so it suffices to learn $\mathcal{R}(.,.)$ as a $(|\mathcal{S}| \times |\mathcal{A}|)$-sized discrete lookup table. The problem definition above takes inspiration from inverse reinforcement learning (Russell 1998), where an agent tries to learn a reward function from demonstrations. Without loss of generality, we will assume that γ is simply a given hyperparameter in the presentation in this section. The impact of exact choices of gamma will be discussed later in the evaluation section.

6.4 Predictive Modelling based on Demonstrations

It is trivial to show that past demonstrations will have a large overlap in the set of nodes in their expanded trees. By exploiting this overlap, a predictive model can be constructed. The

model is supposed to predict the performance numbers (speedup / execution time) of a particular benchmark given a set of inlining decisions.

6.4.1 Preprocessing Execution Times

The raw execution times are first preprocessed so that they are roughly centered around zero, whereby a positive number denotes an improvement over some reference, and a negative number denotes a performance degradation. Some choice of preprocessing functions are as follows:

- **raw speedup over reference** $-\frac{time-time_{ref}}{time_{ref}}$. The most intuitive value representation of the execution time. A good reason to use this benefit function is that it is easily comprehensible by a human.
- **tanh speedup over reference** $tanh(\frac{\alpha\times(time-time_{ref})}{time_{ref}})$ Attempts to capture the non-linearity of benefit with respect to speedup. For example, an inlining decision that results in a 70% speedup is most likely comparable to one that yields a 35% speedup, that is, in both cases, we *will* want to inline them, and the amount of speedup is no longer relevant.
- **logarithm of difference reference** $log(\frac{time_{ref}}{time})$ - The argument is the relationship between node speedups and their children might be multiplicative. Consequently, in the logarithm space, they are additive.

t_{ref} corresponds a reasonably selected reference execution time. We have used the baseline time (execution time of the default compiler) and the mean execution times as references in our experiments. The output of preprocessing the execution time is the known *target values* of a given expanded tree in a benchmark.

6.4.2 Estimating Local Reward, $\mathcal{R}$, with Linear Regression

For a set of M demonstrations, each expressed in terms of an expanded tree, its *predicted value* can be expressed as a linear combination of the local rewards at every N observed node-action pairs across the entire tree:

$$\hat{V}(tree) = \boldsymbol{a}^T \boldsymbol{r} \tag{6.4}$$

where $\mathbf{r} \in \mathbb{R}^N$ is the localised reward for every node in the expanded tree, $\mathbf{a} \in \mathbb{R}^N$, a_i is a term defining the weight of node i to the tree's value in that particular execution.

Different inlining decisions can be composed of different trajectories, so it is possible for $\boldsymbol{a}$ to contain zero entries (in fact, it can sometimes be very sparse). The $\mathbf{a}$ for a single execution is called a demonstration vector. Entries in the demonstration vector can be constructed by $a_i = \gamma^{depth(i)}$ (When $\gamma = 1$, $\mathbf{a}$ is a binary vector). By stacking demonstration vectors together, a demonstration matrix A can be obtained. An illustration of the construction of the A is depicted in figure 6.6.

Given a demonstration matrix A and some local rewards vector $\mathbf{r}$, the *predicted value* can be expressed as $\hat{\mathbf{v}} = A\boldsymbol{r}$. It then follows that the *local rewards* of node-action pairs in the expanded tree can be solved by finding $\mathbf{r}$ that minimises the the mean squared error (MSE) of the predictive model:

$$J(\mathbf{r}) = \frac{1}{M}(A\boldsymbol{r} - \mathbf{v})^T(A\boldsymbol{r} - \mathbf{v}) \tag{6.5}$$

The main problem with this approach is that without explicit care during exploration, A is likely to be a underrepresentation of this problem ($rank(A) < min(M, N)$). Malformed data of this form is, unfortunately, abundant in our dataset. An illustration of of this problem is shown in figure 6.7. The most likely scenario for such iterative compilation is that the problem will be underrepresented - many of the state-action pairs in the expanded tree will not be observed. Observing all nodes requires an excessive number of iterations in iterative compilation.

One way to overcome this problem is to use *L2 regularisation* (Ng 2004). The heuristic being used in the formulation is that localised rewards ought to be as small as possible. This changes the mean squared error formulation to the following:

$$J(\mathbf{r};\lambda) = \frac{1}{N}(A\boldsymbol{r} - b)^T(A\boldsymbol{r} - b) + \lambda \boldsymbol{r}^T\boldsymbol{r} \tag{6.6}$$

where λ is a hyperparameter. This bears similar form to the ridge regression. An analytical solution for an expression in this form exists:

$$\mathbf{r}^* = (A^TA + \lambda MI)^{-1}(A^Tb) \tag{6.7}$$

This method works well when the posed problem is a low-pass filter of the vector space x to the target. In the case of the problem at hand, this is true, as it takes the combination of various rewards due to inlining decisions, most of which are noisy.

Another form of regularisation that can be used is *L1 regularisation* (Ng 2004), which results *sparse* weights. A rationale for sparse weights it that most inlining decisions usually do not matter, but can result in significant performance differences when they do.

The minimisation objective with L1 regularisation transformed into lasso regression:

$$J(\mathbf{r};\lambda) = \frac{1}{N}(A\boldsymbol{r} - b)^T(A\boldsymbol{r} - b) + \lambda \sum_i |r_i| \tag{6.8}$$

where λ is a hyperparameter. With L1 regularisation, the minimisation objective no longer has an analytical solution, and needs to be solved iteratively.

Some combination of ridge regression and lasso regression can be used as well to model the problem with elastic net. While the flexibility will probably allow space for better hyperparameters, there remains, there is some open-ended questions about the procedure to select hyperparameters. The usage of elastic net will not be investigated in this thesis.

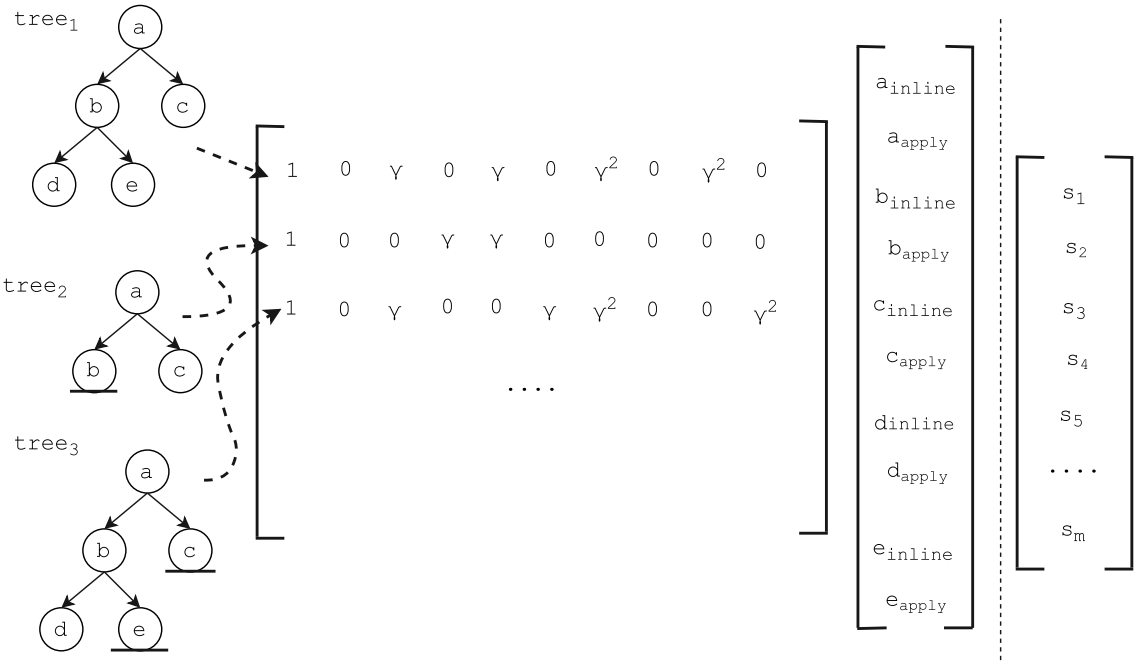


Figure 6.6: An example depicting the construction of the reward-assignment problem in terms system of equations in matrix form. The column vector of $[s_1, s_2, ..., s_m]^T$ corresponds to the target benefits, ie: the minimisation targets. Note that it is possible for inlined function calls to be leaf nodes in the inlining tree.

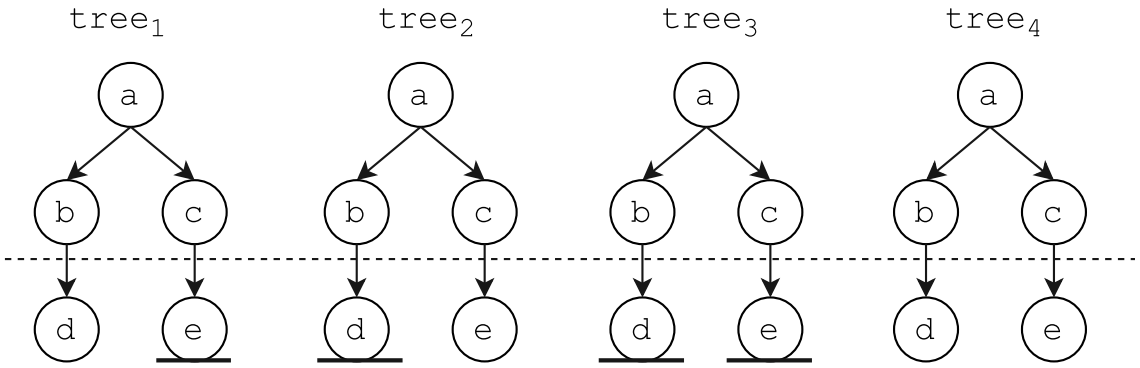


Figure 6.7: An under-representation of the tree benefit relation. The systems of equation cannot solve reward values for $a, \overline{a}, b, \overline{b}, c$ and $\overline{c}$.

6.5 Evaluation

6.5.1 Generating Optimal Decisions with the Predictive Model

To evaluate the effectiveness of this reward-assignment model, an optimal inlining tree is constructed using the learnt predictive model. `GenOptimalTree` as described below returns a $(\mathcal{V} \times \mathbb{R})$. pair, denoting the optimal subtree and it's value.

Input: The root of the expanded tree
Output: $(\mathcal{V} * \mathbb{R})$ corresponding to the optimal expanded tree and its predicted value.
Function `GenOptimal(`s`)`**:**

if $is\_decl(s)$ **then**
  return $(0, Decl\_node(s, \{snd(\texttt{GenOptimal}(c)) \mid c \in \mathcal{C}(s)\}))$
end

$children \leftarrow \{\texttt{GenOptimal}(c) \mid c \in \mathcal{C}(s)\}$
$value_{children} \leftarrow \sum_{(v,subtree) \in children} v$
$value_{inline} \leftarrow \mathcal{R}(s, Inline) + (\gamma \times value_{children})$

if $value_{inline} > \mathcal{R}(s, Apply)$ **then**
  return $(value_{inline}, Inlined\_node(s, children))$
else
  return $(\mathcal{R}(s, Apply), Apply\_node(s))$
end

Algorithm 3: Generating the most optimal expanded tree using $\mathcal{R}$ learnt via predictive modelling

To compare the effectiveness of this model, the "optimal" expanded tree can be used to generate a set of "optimal" inlining decisions - the resulting performance ought to be compared with the observed data. Ideally, the generated "optimal" tree ought to be at least as good as the best observation, but should pick up at least 50% of the observed minima's performance (ie: if the minima is 10% speedup, predictive modelling should be at least 5% speedup.). There are three hyperparameters when formulating a model, the decay factor γ, the choice of preprocessing function f_{exec} and the regularisation parameter λ. The best set of general-purpose hyperparameters that fits all benchmarks is selected using the following formulation, are chosen based on how *well* it reconstructs the demonstration set's observed minima[2]:

$$h_{general} = \arg\min_{h \in \mathcal{H}} \left(\sum_{b \in \mathcal{B}} (time_h^{(b)} - time_{baseline}^{(b)}) / (time_{minima}^{(b)} - time_{baseline}^{(b)}) \right) \tag{6.9}$$

[2]Alternatively, we could have chosen the set of hyperparams that attains the maximum speedup

The custom-chosen hyperparameter for a benchmark b is defined as:

$$h_*^{(b)} = \arg\min_{h \in \mathcal{H}} time_h^{(b)} \tag{6.10}$$

The custom-chosen hyperparameter will be referenced as H_*. For completeness, $H_* \in \mathcal{B} \to \mathcal{H}$ can be expressed as a function, as follows:

$$H_*(b) = h_*^{(b)} \tag{6.11}$$

6.5.2 Performance

An exhaustive table of all the results obtained from predictive modelling is depicted in table E.1 in the appendix. A useful illustration of performance of the chosen $h_{general}$ for both L1 and L2 regularisation is shown in figure 6.8 and 6.9. The two graphs illustrates the reconstruction quality and attained speedup over baseline. The reconstruction quality measures how well the predictive model captures improvement in the demonstrations. 0 indicates that it matches the performance of the observed minima, whereas 1 indicates it is only as good as the baseline. It is possible that these numbers lie outside the given range.

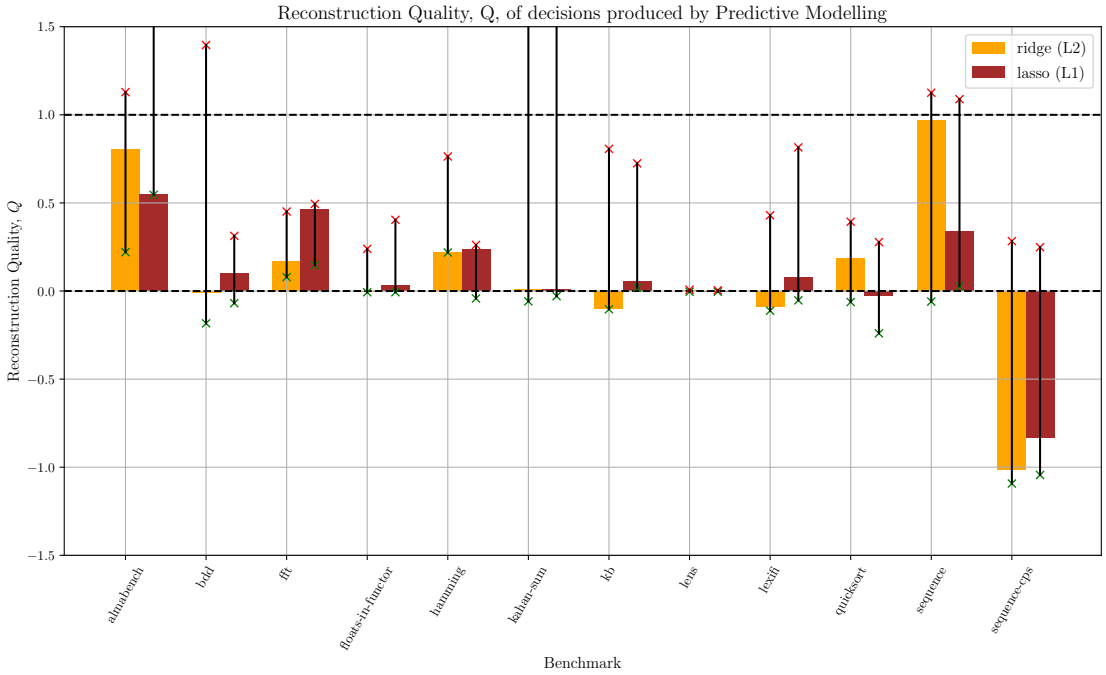


Figure 6.8: Reconstruction quality of predictive models on benchmarks (lower better), with $h_{general}$ with both L1 and L2 regularisation. The omitted upper-bound reconstruction errors are 2.21 (almabench L1), 2.07 (kahan-sum L2) and 3.67 (kahan-sum L1). The lines across bars indicates the range of values obtained over the sampled hyperparameter space. The green point refers to h_*, whereas the red point corresponds to the worst hyperparameter values.

H^* (with the exception of `almabench` with L1 regularisation) managed to reconstruct at least 20% of the speed gain obtained from the observed global minima. This suggest that the predictive model can fundamentally assign sensible values in both regularisation schemes, suggesting that both the assumption about sparse rewards and small local rewards are empirically valid. However, there is no immediately obvious similarities amongst the hyperparameters defined by H^* (an exhaustive list is in table E.3 and E.2 in the appendix). This suggests that most problems have very different performance characteristics and requirements in extreme optimality, but without any further suggestive evidence, it might just be an indication of overfitting.

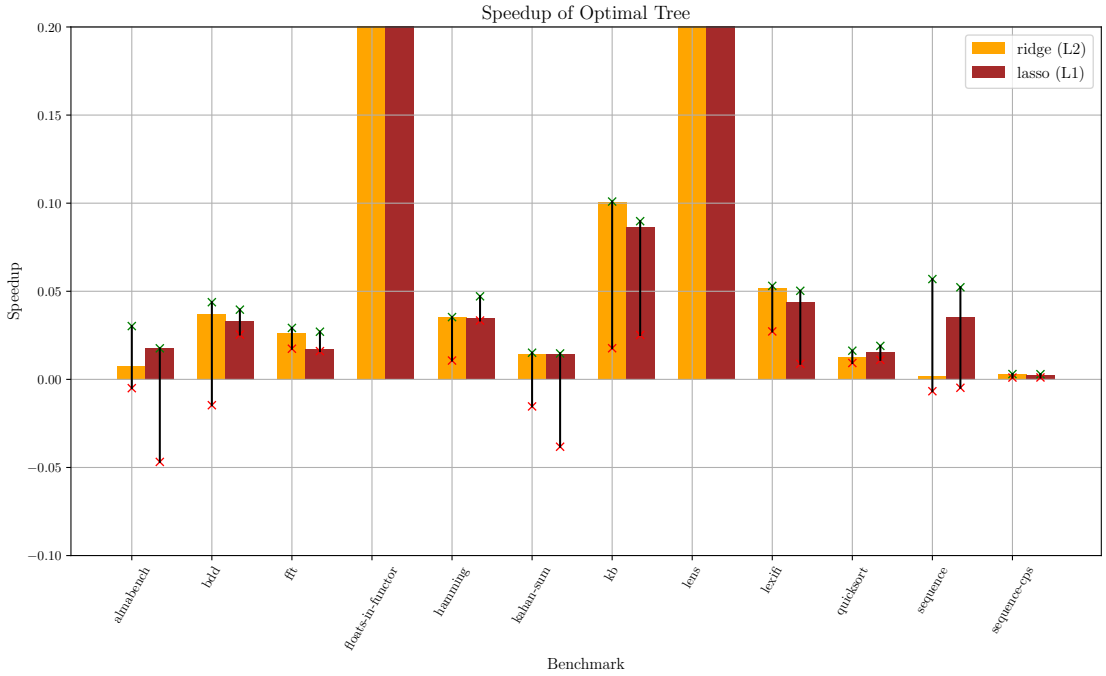


Figure 6.9: Speedup of predictive-models' decisions on benchmark (higher better) of $h_{general}$. Speedup due to `lens` and `floats-in-functor` are close to 50%. The lines across bars indicates the range of values obtained over the sampled hyperparameter space. The green point refers to h_*, whereas the red point corresponds to the worst hyperparameter values.

The general purpose hyperparameters manage to capture most of the performance yield by the benchmark in two outlying cases, namely `lens` and `floats-in-functor`, where roughly 50% speedup is observed. This suggests that the predictive-model can work well in the existence of extreme performance outliers. In the remaining benchmarks, the general-purpose hyperparameters are within 15% reconstruction quality from the optimal hyperparameter set in 7 our of 10 times, in both L1 and L2. In the other examples, they never perform worse than the default set of decisions ($Q < 1.0 \wedge speedup > 0$), suggesting that the notion of a "general-purpose" hyperparameter set is empirically backable.

Interestingly, in `kb` and `lexifi`, $h_{general}$ and h^*_{exp} both managed to predict a set of decisions that are 5-10% better than those observed from iterative compilation. This can be a result of the predictive model successfully combining the inlining decisions from several unrelated call sites. While the same can be said for `sequence-cps`, the performance gains is minimal.

6.5.3 Reward Statistics - What did the Predictive Model Learn?

NB: The plots below are generated using data obtained from h^*. Similar statistics can be derived from $h_{general}$

Distribution over (Sub-)Tree Values

Figures 6.10 and 6.11 shows the histogram of optimal sub-tree values $V*$ of the optimal-tree of $h_{general}$ using both L1 and L2 regularisation. In both cases, these values are empirically form an log-normal distribution.

There is a significantly higher-number of zero rewards in the L1 variant, as opposed to the L2 one. This is expected due to the nature of L1 regularisation. Interestingly, the mean value sub-tree values is roughly a magnitude smaller than those in L1, corresponding roughly to the magnitude of the difference in number of zero entries.

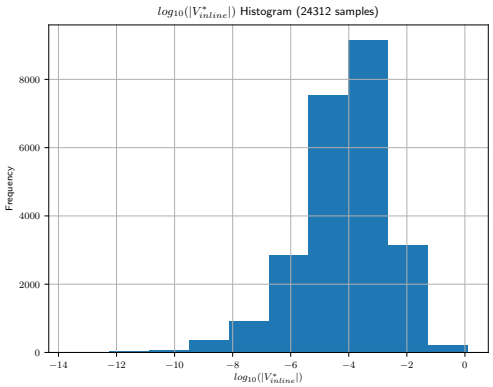


Figure 6.10: Histogram of logarithm of total value with L2 regularisation.

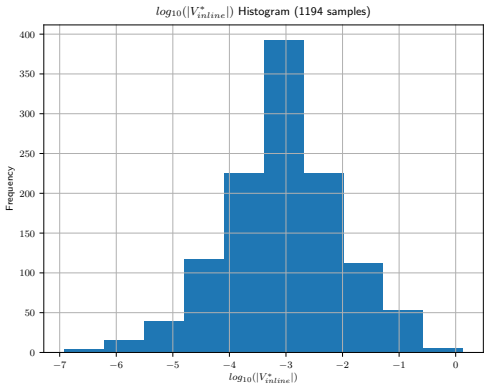


Figure 6.11: Histogram of logarithm of total value with L1 regularisation.

A simple conclusion can be drawn from this observation: singular call sites, that is call sites where the dataset contains only demonstration of one inlining decision, are assigned substantially different values. In L2 regularisation, these decisions are assigned a very small value, like 10^{-4} whereas in L1 regularisation, these decisions are assigned zero (or a simple a negligibly small value, due to numerical errors when optimising for a solution). This happens because $(r_1^2+r_2^2) \leq (r_1+r_2)^2$, ie: the solution will favour spreading out a weight across multiple small weights rather than concentrating it in a single variable. As a result of this distinction, tree-value V^* computed, in the case of L2 regularisation, will have incorrectly accounted for the value noise from the singular call sites. L1 regularisation does not suffer from this problem, as singular call sites are usually just assigned a zero local reward. This distinction is best illustrated by comparing the graphs illustrating the learnt values in bdd using L2 regularisation (figure 6.12) and L1 regularisation (figure 6.13) The abundance of singular call sites. The two plots includes *all* non-zero data-points across the entire dataset. L1 learnt 118 non-zero values, whereas L2 learnt 6347 non-zero values reward values. Given that in our formulation of `bdd`, only 630 out of of the 6347 nodes are non singular, it is evident that L2 has assigned non-trivial arbitrary values to many of the singular call sites.

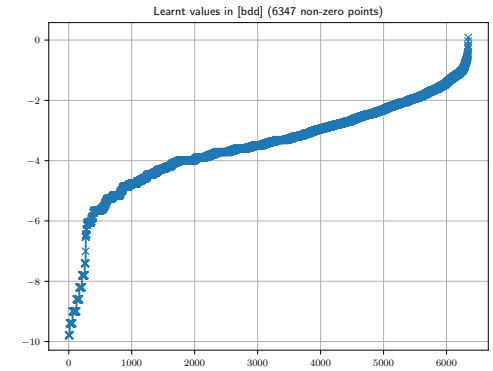


Figure 6.12: Learnt local rewards with L2 regularisation

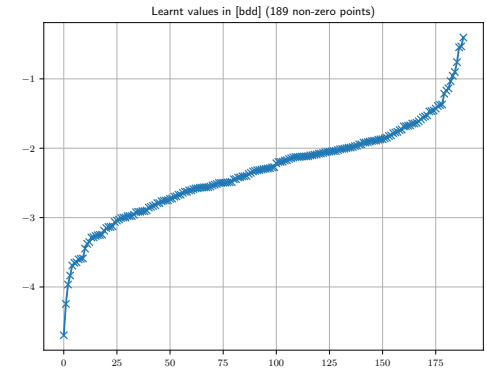


Figure 6.13: Learnt local rewards with L1 regularisation

V^* vs $\mathcal{R}_{apply}$ in L1 Regularisation

When the long magnitudes of sub-tree value is plotted against the log magnitude of termination reward, four main clusters of data-points are formed, as shown in figure 6.14. The four clusters are: (1) trivial inline and termination reward (bottom-left); (2) trivial inline, but non-trivial termination reward (top-left); (3) non-trivial inline, but trivial termination reward (bottom-right);

and (4) non-trivial inline and termination reward (top-right).

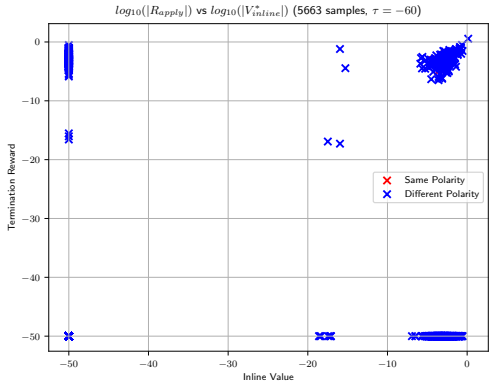


Figure 6.14: Sparse rewards in L1, resulting in obvious decisions. Values that are $< 10^{-50}$ are represented as 10^{-50}

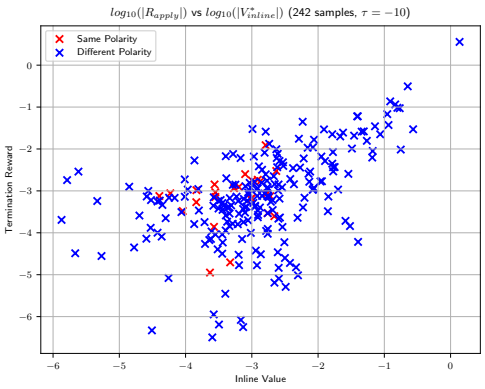


Figure 6.15: Zooming into the top-right cluster in the plot shown at the left (figure 6.14).

In the first cluster, it is likely that the inlining decision does not matter. In clusters (2) and (3), the reward-assignment model has figured out that one action is without a doubt, significantly better / worse than the other. Cluster (4) presents a possible source of ambiguity, in which the magnitudes of the values or somewhat correlated. Interestingly enough, when zoomed into cluster (4), as illustrated in figure 6.15, most of the points actually turn out to have different polarity (ie: $x > 0 \wedge y < 0$ or vice versa, only 10% of the points in cluster (4) have the same polarity), suggesting that the model did in fact model a significant predictive different between *Inline* and *Apply*. A consequence of this is that the predictive model constructed by lasso regularisation will almost certainly capture obvious difference between decisions certainly. However, the lack of continuity (since it is mostly deciding a value of zero) implies that the model might struggle to work with inlining strategies that is composed of a series of well-coordinated inlining decisions.

V^* vs $\mathcal{R}_{apply}$ in L2 Regularisation

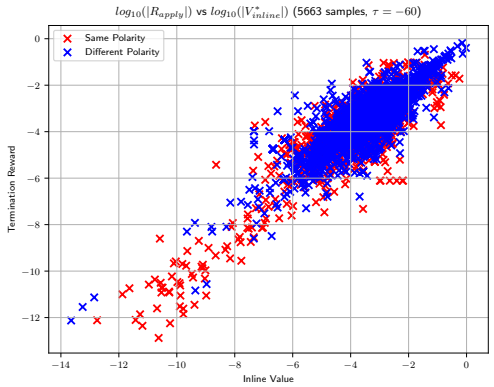


Figure 6.16: Correlation between the logarithm of magnitudes sub-tree values and termination reward with a gradient that looks to be approximately 1. A lot of the red markers are obsucated by the plot.

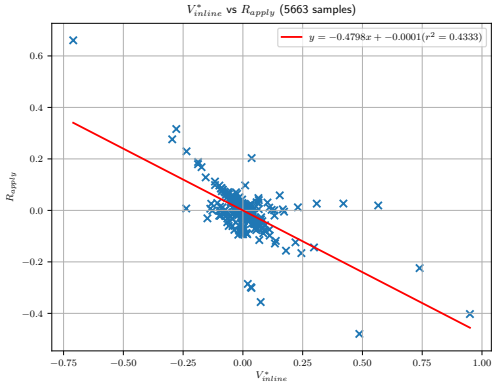


Figure 6.17: Negative correlation between sub-tree values and termination compensation.

A plot of the correlation between the magnitudes of sub-tree values in log space is show in figure 6.16. There is no clear notion of cluster separation in this plot - unlike those in L1. In the given

plot, 25% of the points are red points, that is 75% of the points have different polarity. The interesting observation here is that the magnitudes of $\mathcal{R}_{apply}$ and V^* do not differ significantly.

As shown in figure 6.17, the two values exhibit a negative correlation. The model prefers to assign counter-acting values $(x/2, -x/2)$ to the two node-action values, rather than $(0, -x)$ or $(x, 0)$. This results on a more continuous distribution of rewards along the path on an expanded tree. For this reason, we suspect that this regularisation will work better for inlining strategies that are composed of a series of well-coordinated inlining decisions.

6.5.4 Tuning λ in Lasso Regression

Given that the fundamental reason to use lasso regression is to learn sparse rewards, how sparse should it be? By increasing λ, the sparsity of the rewards will increase to a point where the the model simply assigns zero to all local rewards. By taking inspiration from Ng et al. (2000), We initially suspect that (1) the transition from a zero local rewards vector to a non-zero vector will form a unit step function when varying λ logarithmically, and (2) maximum performance-per-non-zero-weight ratio by decreasing λ until the point right after the weight passes a constant threshold $\tau \approx 10$. The assumption is that there is unlikely to be that many nodes that results in massive performance increases. By devising a means to automatically tune λ, we reduce the number of benchmarks that needed to be execute to find h^*_{exp}

To test this assumption, we performed a series of experiment varying λ in logarithm steps whilst keeping other hyperparameter constants in `lexifi` and `bdd`. A representative test case is shown in figure 6.18. It is evident that the best speedup is not attained when λ turns from zero to non-zero. As a matter of fact there is no clear cut-off point - the transition is not a unit step function, this refutes our first suspicion. Furthermore, by inspecting the bottom-most graph in figure 6.18, it is not clear whether the notion of speed up per non-zero weight bears any meaning at all! The number decreases so quickly to zero as the number of weights increases that the numbers are no longer comparable. A pathological test-case challenging our second assumption is shown in figure 6.19 - the test-case with the candidate with the best performance per non-zero weight acheives the lowest speedup. As a result of this, for most practical purposes, no reasonable conclusion can be drawn on smart-tuning λ while retaining performance.

In the process of figuring out an algorithm to automatically tune regularisation parameter, we found out a more obvious and simpler way - that is to use some of the executions as a validation set. This is a valid thing to do since the goal of predictive modelling is to actually predict performance. In figures 6.18 and 6.19, the green dotted line denotes the λ where the validation error is the lowest (equivalently, highest coefficient of determination (r^2)). From the set of experiments that we have carried out, using a validation set finds a parameter that achieves a speedup within 1% of the best speedup.

6.5.5 How to Select a Reward Assignment?

The intended goal of going through the hassle of designing a predictive model for function inlining is so that we can quantify the values of inlining decisions at function call sites throughout the program. Instead of having a single reward-assignment, we now have several ($h_{general}$ with L1, H_* with L1, $h_{general}$ with L2 and H_* with L2). There is fundamentally just no justifiable reason to select one hyperparameter set over the other - and there is no reason why we should not investigate the effectiveness of all 4 reward assignments in designing an inlining policy. On top of that, we have selected two set of hyperparameters that we believe should have worked well intuitionally, that is h_{hand} for L1 is $\gamma = 1.0, f_{preprocess} = \text{tanh\_speedup\_over\_baseline}$ and one for L2, $\gamma = 0.4, \lambda = 0.05, f_{preprocess} = \text{log\_speedup\_over\_mean}$. The intuition behind the former is that the sparsity of rewards in L1 warrants a sum over the entire tree without any decay factor, since stability is less of an issue. $f_{preprocess}$ is almost arbitrarily chosen in both cases, but picked such that it encapsulates that squeezing effect inlining benefits at large magnitudes. The general rule of thumb in science is not to trust human intuition to make final decisions, but it will be foolish not to trust it at all! (Kahneman & Egan 2011).

As a result, we now have 6 possible reward assignments.

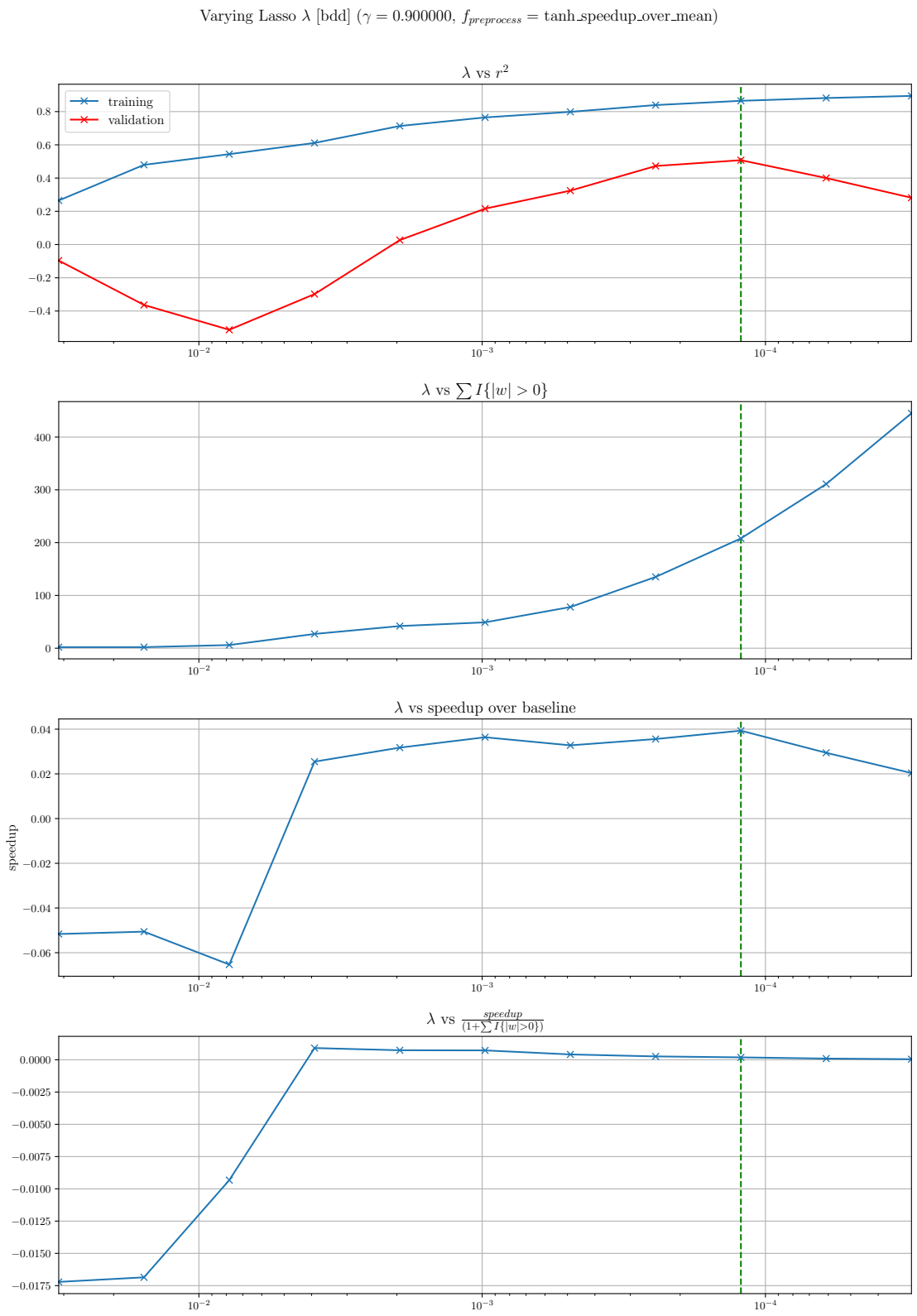


Figure 6.18: An experiment varying λ in lasso regression `bdd`

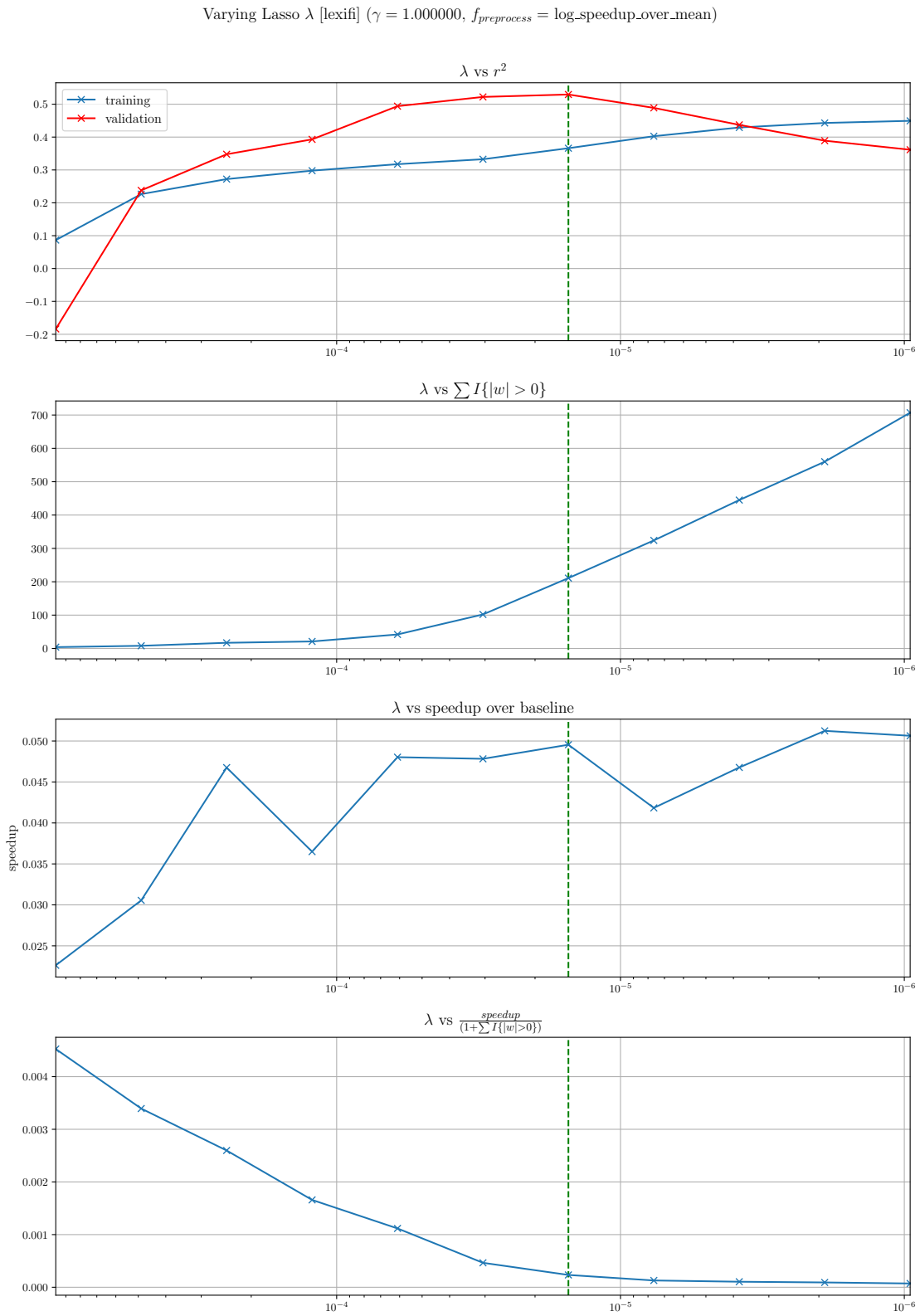


Figure 6.19: An experiment varying λ in lasso regression in `lexifi`

6.6 Summary

In this chapter, we have discussed methods and algorithms to turn a sets of inlining tree and execution times for a given benchmark. Several ideas have been proposed in this chapter:

- the *incremental tree-space exploration* problem has been defined
- the *expanded tree* data structure has been proposed as an alternative representation of the inlining tree where the space of inlining trees are wonderful.
- a *predictive modelling* methodology based on linear regression to learn optimal inlining decisions from demonstration from a program benchmark.
- *assigning numerical values* to inlining decisions at function call sites throughout the program, by expressing *expanded tree*-space exploration as an incremental tree-space exploration problem, and solving it using demonstration-based predictive modelling.

The numerical value assignments will be used as a fundamental building block for inlining policies in the following chapter.

Chapter 7

Learning Data-Guided Inlining Policies

In the previous chapters, a series of methods from transforming microbenchmarks into sets of function call sites (or inlining decisions) with their associated long-term value $V*$, local inline reward $\mathcal{R}_{inline}$ and termination reward $\mathcal{R}_{apply}$ compensation have been presented. This chapter discusses how machine-learning can be employed to learn an inlining policy based on the said data. A thorough evaluation on the effectiveness of the methods is deferred to the next chapter, as the results of this chapter is very much the evaluation of the entire project itself. Compared to other chapters, this chapter is more "empirical" and lacks theoretical presentation. Most of analysis here is built on top of existing data analysis techniques on our own dataset.

7.1 Constructing Raw Training Data

To collect inlining queries[1], the compiler is modified to dump all inlining queries and their associated inlining trace. This requires recompilation, but does not require the benchmark to be rexecuted (hence, this is bottlenecked by the available benchmark resources).

The result of reward assignment from the previous chapter in the form of an expanded tree tagged with their associated values, local reward and termination reward. Every node in the said tree has an inlining trace, so inlining queries can be mathced with a reward-assignment by matching their inlining traces. By constructing a binary-search tree to perform this lookup, the time complexity for binding $M \times N$ inlining traces across M experiments with N nodes in the expanded tree is upper-bounded by $O(M \log N)$. As proven in 4 - this means of labelling is indeed valid. In a correct implementation, all inlining queries should have an associated reward label in the expanded tree. One slight subtlty is that we assume that the inlining queries with a similar inlining trace are similar enough that an arbitrary choice of amongst the set of inlining queries can be used[2].

A summary of the statistics of the available training data is available is depicated table 7.1. There is a high number of function call sites throughout the program with $\mathcal{R}_{apply}$ - this is expected when function inlining is not possible altogehter, eg: when using first-class function arguments. There are roughly 1199 occurence where information about the rewards cannot be derived - this happens only in two of the experiments, namely `lexifi` and `kb`. This can most likely be attributed to a bug in our implementation. The median depth of these *missing data points* 14-levels deep in the expanded tree, so it is most likely that these decisions should not severely affect the outcome of the learnt policy. The existence of datapoints with *only* inline rewards can be attributed to the incompleteness in exploration and path expansion.

[1] a bin dump of several data structures that can be accessed when making an inlining decision, such as the function declaration body and number of known value bindings.

[2] this not necessarily true - inlining depth will not be the same, for example

No Information about rewards	1199
Both inline reward and termination compensation	4388
Just inline reward	2228
Just termination reward	10026
Total	17841

Table 7.1: The data on function call sites that is available as training data.

7.2 Feature Engineering

As a simple preprocessing step, an inlining query is transformed into a finite feature vector $\mathbf{x} \in \mathbb{R}^N$, as most machine learning algorithms operate on finite vectors. The features are selected such that they can be derived from the function calls in Flambda and the compilation passes' context (called the environment). These features are selected based on (1) suspected relation with inlining decisions and (2) easy to obtained without significant programming effort. An exhaustive list of the used features is provided in table B.1 in the appendix. The features generally falls into one of the following categories:

- **Approximation features** - These features are features about the function body. There are two set of such feature, one that belongs to the inlined function body and one that belongs to the function body. This allows the model to learn features that compares the number of approximation in the original function body and inlined function body.
- **Function declaration features** - These features are features about the nature of the function declaration, without regards to the function body. These features can be useful, especially due to the fact that Flambda treats function argument variables as free variables.
- **Argument features** - information about the tag arguments being passed into the function. This allows for exploitation on inlining that generics. It was arbitrarily chosen to have 7 arguments subject to these argument decomposition. These features are discretised to boolean features with one-hot encoding, so an argument is represented by 14 features.
- **Environment features** - information about the caller and its surrounding context, such as the depth of function inlining that has been performed prior to unveiling the said function call site.
- **Flambda Features** - these features are directly related to performance implications, namely the number of removed primitive operations, removed indirect function calls and removed block allocations
- **Callee body features** - these features have almost nothing directly to do with function inlining. They are simply a form proxy for categorising the structure of the function body.

A features that is not included (perhaps surprisingly) is whether the function is a recursive call. In programs that statically compile all function calls, it is definitely possible to know a function call candidate ahead of time (function pointers are never inlined). In the case of FP languages, it is possible to have the function itself passed as an argument, making checks for recursive functions tricky.

The result of this feature extraction is the generation of two types of features - numerical ones and boolean ones. After collecting these features from the inlining queries available in the dataset, a simple process is used to remove useless features, ie: features that are always constant (boolean features) or have an extremely low variance (numeric features). This step reduces 64 numeric features to 228 and 115 boolean features to 51. The existence of these and constant or low-variance features is due to the limitation of the small sets of benchmarks used for training. Prior to any analysis and training, the numerical features are normalised, a standard data preprocessing step in data analysis:

$$\mu_i = \frac{1}{M} \sum_{\mathbf{x}} x_i \tag{7.1}$$

$$\sigma_i = \frac{1}{M} \sum_{\mathbf{x}} (x_i - \mu_i)^2 \tag{7.2}$$

$$z_i = \frac{x_i - \mu_i}{\sigma_i} \tag{7.3}$$

The resultant feature vector is the concatenation of $\mathbf{z}$ and the vector of non-constant boolean features, $\mathbf{x} \in \mathbb{R}^{79}$

7.3 Generating Classification Labels

To curate training data for an inlining policy, labels are generated from the output of the reward-assignment process presented in the previous chapter. Note that as discussed in the previous chapter, we have different number of

$$f(x;\tau) = \begin{cases} 0 & |x| < \tau \\ x & otherwise \end{cases}$$

where τ is a number for denoting negligible reward or tree values. The choice of number is sensitive to the optimisation methodology used for lasso regularisation. The classification labels are then generate as follows:

$$y(v;h,\tau) = \begin{cases} IDontKnow & f(V^*(v;\tau)) = f(\mathcal{R}_{apply}(v;\tau)) \\ Inline & f(V^*(v;\tau)) > f(\mathcal{R}_{apply}(v;\tau)) \\ Apply & f(V^*(v;\tau)) < f(\mathcal{R}_{apply}(v;\tau)) \end{cases}$$

7.3.1 Using Lasso Reward Assignment

As we have seen in the previous section, most rewards learnt in the lasso-based regression model is are zero values. Extremely small values ocassional exist due to precision error, so care has to be taken to different small rewards and numbers that are practically zero. We have found that $\tau = 10^{-8}$ to be a representative threshold to differentiate very small values and extremely zero-like values. The distribution of the class labels, after generating labels from L1 regularisation is as follows:

hyperparams	I don't know	Inline	Apply
$h_{general}$	0.947	0.031	0.022
h_{hand}	0.911	0.057	0.032
H_*	0.872	0.082	0.045

The distribution is all cases highly skewed - there is a high proportion of "I don't know" training examples compared to anything else. This is consistent with the assumption of the L1-based reward assignment model - most inlining decisions don't matter.

7.3.2 Using Ridge Reward Assignment

Selecting τ for ridge regression is much trickier - there is no clear-cut off point of zero point between zero-values and very small values. This is clearly illustrated in figure 7.1. The proportion of training examples labelled as *Inline* and *Apply* is roughly the same. The proportion of "I don't know" exhibits a linear relationship with respect to $log(\tau)$. Increasing τ should, based on the proportion of 'I don't know' examples in the training set, yield a more stable inlining policy. A stable inlining policy is one that makes more decisions more conservatively[3], hence, will have less performance variance when benchmarked on an unseen set of benchmarks. This can be validated by measuring the amount of variance of performance in the test benchmark set with respect to τ.

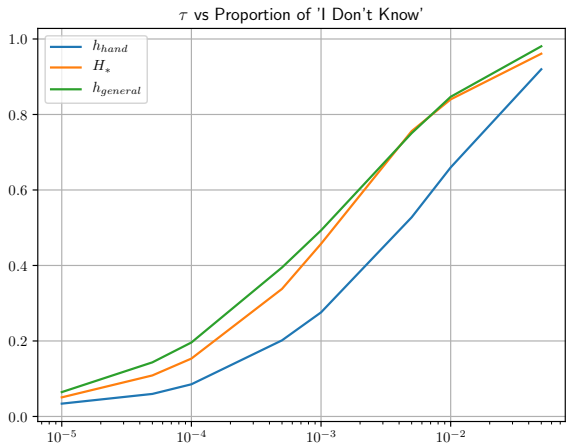


Figure 7.1: Proportion of I don't know entries in H_* (L2) whilist varying τ

Unlike in lasso reward assignment, there is no way to magically select τ without thoroughly benchmarking performance of the set of inlining decisions made by the model. In other words, for every set of reward learnt via ridge regression, we have D possible ways to label them, where D is the number of number of τ that we are willing to try.

7.4 Visualising the Feature Space

Figure 7.2 shows the normalised covariance matrix between feature vectors throughout the training set (Note: this is NOT the covariance matrix between features). The normalised covariance matrix computed by taking the number of

divided by the :

$$\mathbf{z} = \mathbf{x} - \bar{\mathbf{x}} \tag{7.6}$$

$$\hat{\mathbf{z}} = \frac{\mathbf{z}}{|\mathbf{z}|_2} \tag{7.7}$$

$$\hat{\sigma} = \hat{Z}\hat{Z}^T \tag{7.8}$$

where $\hat{Z} = [\mathbf{z}(1), \mathbf{z}(2), \mathbf{z}(3), ..., \mathbf{z}(n)]^T$ is a matrix where each column correpsonds to a unit vector representation of the feature vector. From the occasitional sections of 7.2 bright yellow areas, it is likely that the training set contains very correlated / similar training examples. To visualise the distribution of number of highly correlated examples, a histogram of the number of correlated features a. In other words, visualise a histogram of $f(i) = \sum_j \hat{sigma}_{i,j}$ for $i = 1, 2, ..., N$. The said

[3] what does it even mean to be conservative here? Being conservative with inlining usually means *not* inlining, but in this case, the model falls back to the default inlining heuristics.

histogram is plotted and depicted in figure 7.3. The peak at 1 indicates that the most common occurence is that a feature vector is highly-correlated with itself. As a matter of fact, about 25% of the feature vectors have at most a single repetition in the training set, the third quartile occurs at around 50 highly correlated examples. All this indicates that while highly-correlated examples do exist in the training set, but they are not too abundant that machine learning techniques should still work as expected.

It is possible to remove reptitions of highly correlated examples, which has a worst-case time complexity of $O(M^2N)$. This is perfectly tractable for the classification problem at hand. However, we have chose *not* to remove these training examples, as they are representative of the set of inlining decisions that can occur when making inlining decisions. By inspecting the data, amongst the functions that have many highly-correlated counter parts are `Format.printf`-like functions and its descendants, in addition to recursive calls. While it is possible that these examples are essentially "repeated" noise in the dataset, they *do in fact* show up often in OCaml code. It makes sense only to have an inlining policy pick up its behaviour. The repeated feature vector due to recursive calls might imply that the set of chosen features might not be good enough to differentiate different function call sites. However, due to time constraints of this project, we do not attempt to try investigating a better set of features.

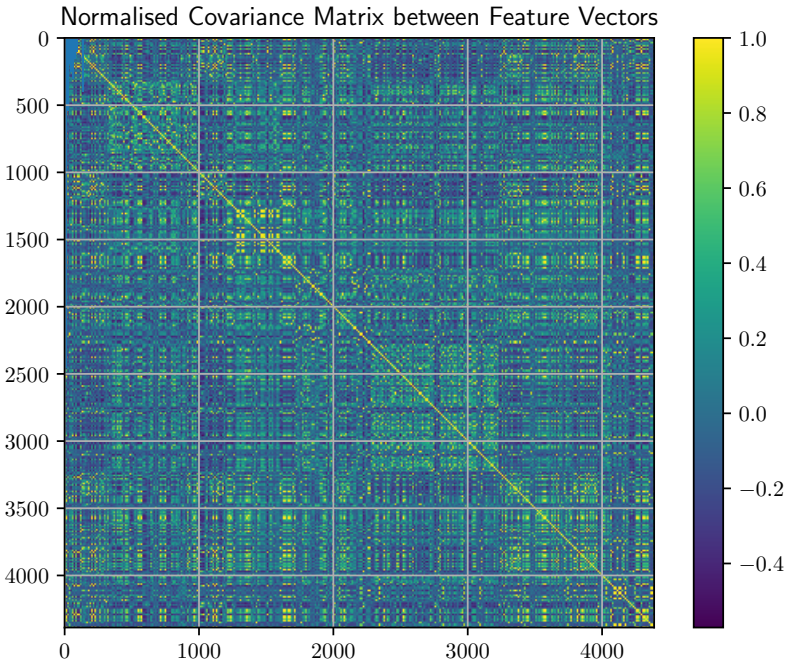


Figure 7.2: Normalised covariance matrix between training examples.

Typically, two visualise the feature space, we use PCA or LDA to perform dimensionality reduction. However, the existence of outliers makes process makes straightforward visualisation difficult. From training a LDA and visualising it, we do not observe an immediately obvious separation between classes. This suggests that the data might not be linearly classifiable.

7.5 Generating an Inlining Policy with Machine Learning

After feature engineering and label generation, we now have a set of feature-labels groups($\mathbb{R}^N \times L_1, L_2, L_3, ..., L_P$) that can be used to develop an inlining policy, where $\forall i \,.\, L_i \in \{Inline, Apply, IDontKnow\}$

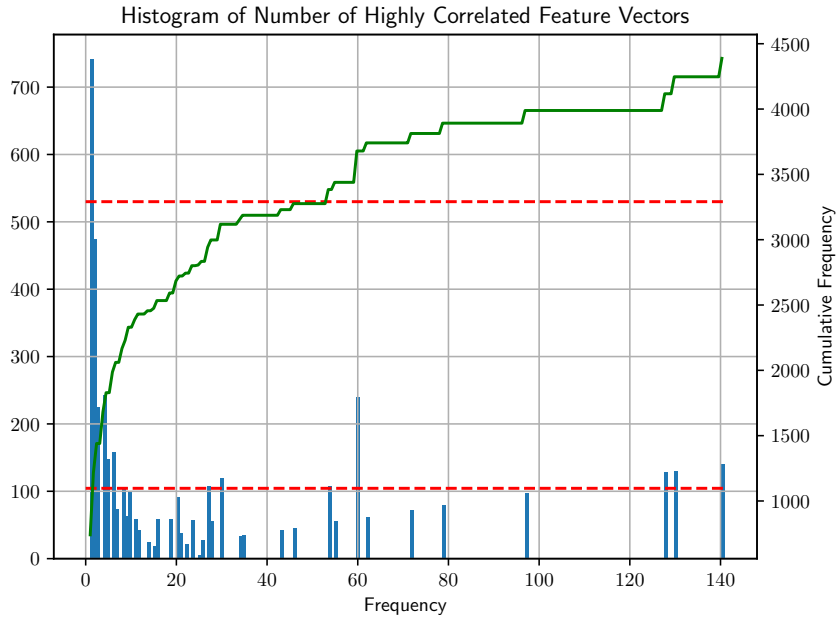


Figure 7.3: Histogram of the number of highly correlated feature vectors and the cummulative frequency plots. Highly-correlated here refers to feature vector whose normalised covariance $>$ 0.999.

7.5.1 Uni-Model Policy

The feature-labels pairs are groups are separated into P feature-label training sets. Each of these training set is uesd to train a classifier backed by an artificial neural network. Each of the neural network has the following hyperparameters[4]

Number of input features	79
Size of hidden layers	{32}
Activiation function	*relu*
Output activation function	*softmax*
Regualarisation	L2 with $\lambda = 0.005$
Loss function	Cross entropy loss function

The result of this process is P classification models, each of which correpsonds to an inlining policy. Each of the model outputs a probability distribution over a set of actions given the features at a function call site. To use this inlining pass, the user compiles P *production* copies of the program, each using one of the P models and benchmark them. The fastest program will then be selected as the output of compilation. WIthout running the programs themselves, it is hard to tell which of the P models will perform well

7.5.2 Chaotic Mixture of Experts (CMoE)

Mixture of Expert (MoE) (Jacobs et al. 1991) is a machine learning technique that combines output of different predictors, each of which trained on separate test cases. MoE is used to divide the problem to a set of homogenous regions, using static partition schemes (such as decision trees (Quinlan 1986)) or adaptive dispatched (such as gating networks (Avnimelech & Intrator 1999)).

Taking inspiration of MoE, we introduce the idea of Chaotic Mixture of Experts (CMoE). That is, instead of using one of the P labels, we use *all* P labels to help make an inlining decision.

[4]NB: The choice to use an artificial neural network and their hyperparameters is arbitrary, and is not backed by a thorough evaluation of the set of possible models to use.

We call this chaotic, because we deliberately train the different experts (classifier) with differing classification labels for the same set of feature vectors.

That is, we train P inlining policies as defined above. However, instead of having to choose a single policy, the output probability distribution over the set of actions from every policy are combined to form a combined probability distribution, which is used to make an inlining decision as above.

$$P_{chaotic}(\hat{(}y) = y|\mathbf{x}) = \frac{1}{Z}\sum_{i \in P} P_i(\hat{y} = y|\mathbf{x}) \tag{7.9}$$

The idea behind MoE is to reduce overfitting, and in the case of CMoE tries to reduce the variance in the output probability distribution. Consider the following scenarios:

- **Misguided Confidence** - It is easy for a single model to be mistakenly confident about its decision without any frame of reference.
- **Unfamiliar Test Cases** - When an inlining policy encounters a novel test case, what should be the output? Assuming the pathological case where the neural network uniformly output one of $[1, 0, 0]$,$[0, 1, 0]$ and $[0, 0, 1]$ in random[5]. The presence of multiple models, all sampling the vector uniformly, will return a uniform output distribution, indicating the policy's uncertainty about the observation.

A pragmatic benefit of this method is that instead of having to choose: Instead of choosing which of the P model to use, we have a single inlining policy combined by P model, eliminating the need to execute P model on a set of benchmarks to choose the optimal policy. In other words, CMoE can be used to construct something closer to a general-purpose static inlining policy that does not require explicit benchmarking.

7.5.3 Compilation Caveat

A slight caveat to compilation: the trained inlining pass often times works well only when the `max-unroll-depth` of a given program is pass a certain threshold. While it is desriable to set this number as high as possible (maximum unroll depth in general implies better performance), setting this number pass a certain threshold will result in a excessively long compilation times, and since OCaml does not explicitly check the size of the IR, it might complete the FLambda pass with a massive IR (with a combination of flags, we had a case where we ended up with Flambda completing its pass, but ended up generating an IR 1GB in size - we never waited for the compilation to terminate, but it was an interesting experiment). Bearing the assumption that if compiling with `inline-max-unroll = x` terminates implies `inline-max-unroll = x - 1` terminates, and compiling with higher `inline-max-unroll` is always better, we can binary search the value of `inline-max-unroll` that does not terminate the program. The (hacky) bash script to perform this binary search is attached in appendix C.

This compilation caveat is taken into account in when we evaluate the model.

7.6 Integrating an Inlining Policy into `ocamlopt`

Integration of inlining policies into the ocaml compiler is done via compiler plugins. OCaml compiler plugins are code that can dynamically loaded using the `DynLink` module in the standard library, similar to shared libraries with a key difference: rather than just loading the symbols, the executes the entry point of the ocaml plugin. For this project, the OCaml compiler being evaluated has been modified to allow plugins to dynamically customise the inlining heuristic used in the inlining Flambda pass. The main benefit of this approach that a new compiler does not have to be constructed everytime the inlining policy is changed. When writing the plugin, it can very quickly become cumbersome to write a different ocaml module everytime a minor change is made to the machine learning model. As a result, this process is streamlined with a simple "model

[5]This is quite true in practice. Neural networks are quite fond of ouputting a vector with a strong peak

compiler" that generates OCaml code given a machine learning model. The flow in developing an inlining policy is illustrated in figure 7.4.

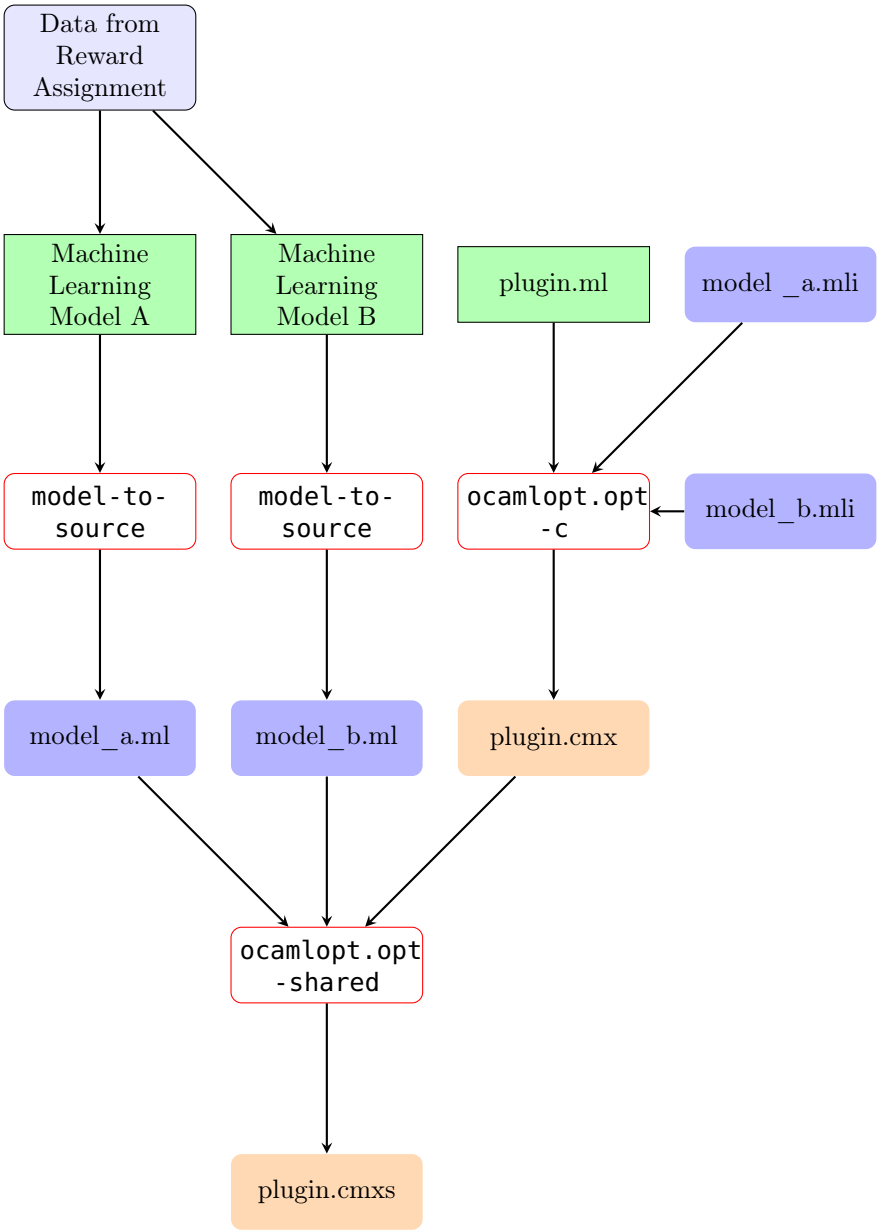


Figure 7.4: The flow for compiling a plugin that is composed of one or more machine learning models. Green components are source code or models designed that are modified during development and purple-ish blue components are code / components that are automatically generated. Orange blocks are compilation artifacts

Chapter 8

Evaluation

Unless otherwise specified, the experimental setup is exactly the same as those used in iterative compilation in the previous section. The exact specifications can be found in appendix D. The main question that the evaluation section here will try to answer is less of "how well did we do?", but rather, "how am I sure I have learnt something?". A unseen set of test benchmark is used for evaluation. The specifics of the benchmarks are described in appendix F.

8.1 Baseline Performance

The closure origin bug. The bug was discovered [1] prior to the beginning of the project. The patch was scheduled to be released in 4.07.0. The bug fix prevents unlimited inline unrolling in a specific corner case. At the beginning of the project, this patch was integrated into the OCaml compiler used in this project. However, there was a bug in the integration, and was only discovered close to the end of the project, when we are trying to perform an evaluation. At that point, the first version of neural-network-based inlining policy plugin has been designed.

Unfortunately, after fixing the bug, the execution results across the training benchmarks with the said inlining policy differs in two versions of the compiler. Specifically, the original observed minima from iterative compilation is attainable in the buggy compiler, but not the fixed one. Fortunately, we found a workaround this problem, that is to artificially increase the value of the `inline-max-unroll` flag in addition to , to reproduce the speedup. As discussed in a section in the previous chapter, increasing the value of this flag beyond a certain threshold will result in excessive compilation times.

To compensate for the bias introduced by this fix, we have to redefine the baseline performance. Bearing the assumption that larger `inline-max-unroll` yields better results, we use the minimum of the time from the old buggy compiler and the fixed compiler using the maximum possible `inline-max-unroll` value as the baseline execution time.

8.2 Uni-Model Inlining Policies

A box plot of the speedup of the inlining policies on the training and test set are shown in figures 8.1 and 8.2 respectively. The key observation here is the the inlining policies do not always result in an improvement in the training benchmarks - as a matter of fact, they sometimes cause performance regressions. While it is not illustrated in the graph, unfortunately, the peak performance attained in every benchmark are not the same inlining policies.

In 8 out of the 12 training benchmarks, the median result of the inlining policy achieves a noticable (>1%) performance improvement over the baseline as defined above. Some interesting performance statistics are as follows:

[1] it was found by the thesis' author! https://github.com/ocaml/ocaml/pull/1340

Figure 8.1: Box plot of performance of programs in training benchmarks.

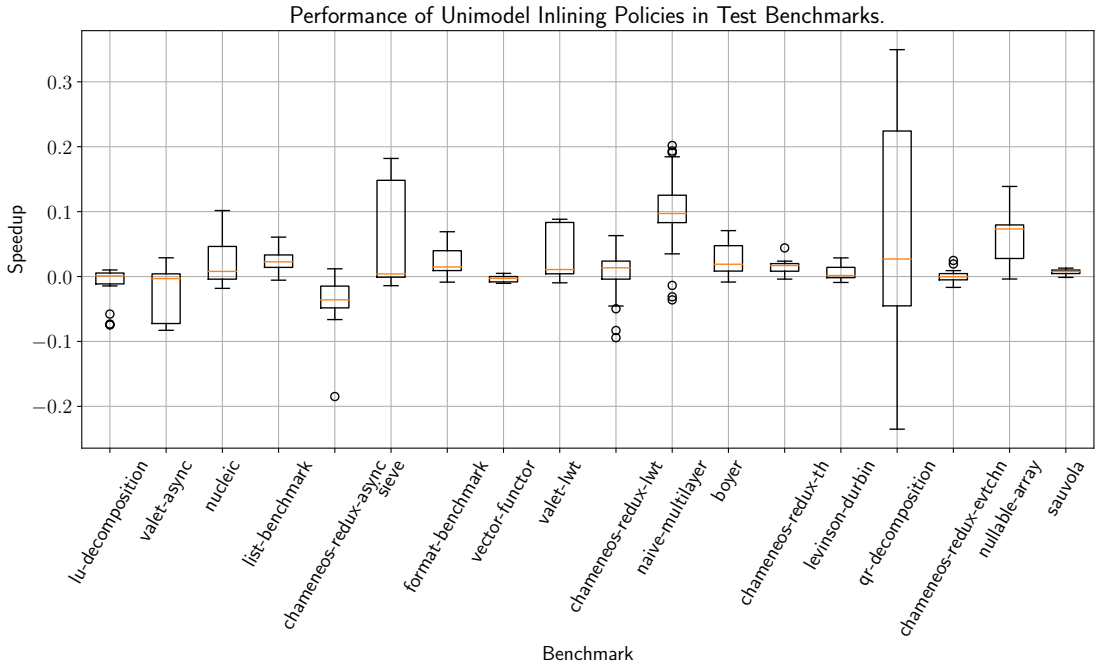


Figure 8.2: Box plot of performance of programs in test benchmarks.

Median of median across training set	1.64%
Median of median across test set	0.93%
Mean of median across training set	9.98%
Mean of median across test set	1.52%
Median of 25-th percentile across test set	-0.13%
Median of 75-th percentile across test set	2.85%

The median numbers are small and not suggestive of any possibility for improvement. From eye-

balling the results in the validation set, the speedup attained from making better inlining decisions far out-weights the set of possible slowdown. A particularly interesting example in the training set is the `kahan-sum` benchmark. While the median speedup is comparable to those original observed in iterative compilation and predictive modelling, the speedup at the 75-th percentile is interesting – a roughly 8% improvement obtained by the machine-learning based inlining policy. As discussed in the earlier section, iterative compilation optimised only the entry-level module, very likely missing out on various inlining opportunities there. The inlining policy here is used over the entire compilation process. On top of suggesting generalisation, this is further evidence that the machine is learning, given that the training set contains floating-point based benchmarks in `float-in-functor` and `fft`, where in the former, a roughly 50% speedup was observed.

8.2.1 Simplifying Model Selection

In a true production scenario, it is a question on how many of this inlining policies should be shipped. The ideal scenario is to ship the policies and recommend inlining models based on the target program being optimised. This requires an additional step of modeling complication. A simpler method is to, given a set of representative benchmark, choose K out of P inlining models that meets a certain criterion. For that purpose we define mean regret, J_{mean} and J_{max} to capture how well a parameter decision captures the known global optimum:

$$J_{mean}(Model_1, Model_2, ..., Model_K) = \frac{1}{|\mathcal{B}|} \sum_{b \in \mathcal{B}} \min_{k \in K} \left(\frac{max_{k'}(speedup_{k'}^{(b)}) - speedup_k^{(b)}}{max_{k'}(speedup_{k'}^{(b)}) - min_{k'}(speedup_k^{(b)})} \right) \tag{8.1}$$

The formulation of max regret is very similar, whereby it has the form $J(...) = \max_{b \in \mathcal{B}} (...)$. The mean regret dictates the average performance loss compared to the optimal scenario, whereas the maximum regret dictates the projected "worst-case" reject. Namely, a choice of models should have J_{mean} as small as possible so long that J_{max} is below a tolerable threshold.

Having define the regret metric, we get back into policy selection. Given P inlining policies, a set of representative benchmark, how do we select K policies such that J_{mean} and J_{max} achieve reasonable values? Finding the optimal choice of K models can be done in time complexity $O(\binom{P}{K}K)$, solving forall $K = 1, 2, ..P$ can be done in $O(2^P P)$ (using an exhaustive search). This obviously requires exponential time, and does not scale well for large P (in our scenario, however, $P = 24$ and $2^P \approx 34 \times 10^6$, a fairly tractable number ofsearches). A greedy heuristic we can use in model selection is to assume that if a model is selected at $K = k$, it will be selected when $K = k+1$. Constructing the for $K = 1, 2, 3, ..., P$ runs in time complexity $O(|\mathcal{B}| \times P)$ but yields an imperfect solution. The rought idea of the algorithm is to incrementally add a model to the model such that at every addition greedily ensures the largest decrease in regret. Figures 8.3 and 8.4 illustrates the change in regret by varying the number of models K based on a greedy allocation policy. A reasonable way to select K is choose K such that $J_{mean} < \phi \wedge J_{max} < \tau$ is minimised, where ϕ and τ are defined by performance and tolerance requirements respectively. The main reason for illustrating this idea is that we can choose *not to* ship all the inlining policy models (or ship all models, and provide the user with a well-tuned default number of models to try). The reduces the compilation time for users without a large number of parallel resources for benchmarking.

8.2.2 Results

To quantify how well the proposed compilation scheme works, we benchmark the results of our compilation compilation pass in the OCaml compiler by providing the grid search on the `max-unroll-depth` and `inline` (the inlining threshold). We set `max-inline-level` as 5 (the default is 1). We performed a grid search of size 9, so make comparable results, we used the greedy allocation method to select 9 uni-model inlining policies. The results are shown in table 8.1. Our model achieves a better result in 8, performs worse than grid search in 3 and performs comparably in 7. We define one model to be better than the other if there is more than a 1% difference in performance.

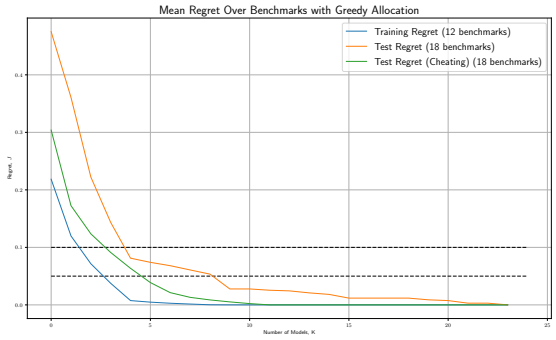


Figure 8.3: Mean regret J_{mean} over benchmarks

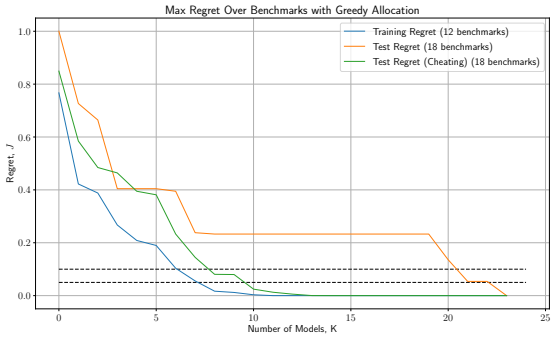


Figure 8.4: Max regret J_{max} over benchmarks

benchmark	Grid Search			Uni-Model Policy	
nucleic	1.062%	15	10	**9.202%**	L2 H_* 0.000050
chameneos-redux-lwt	3.924%	1	25	**6.303%**	L2 H_* 0.000050
list-benchmark	**6.593%**	1	50	5.459%	$h_{general}$ 0.000050
chameneos-redux-async	1.071%	1	50	1.191%	L2 H_* 0.000050
vector-functor	0.013%	5	10	0.405%	L2 h_{hand} 0.000050
valet-lwt	8.476%	1	25	8.630%	$h_{general}$ 0.000100
nullable-array	9.497%	1	25	**13.120%**	L2 h_{hand} 0.001000
naive-multilayer	7.859%	15	25	**19.102%**	L2 $h_{general}$ 0.000050
levinson-durbin	**5.140%**	15	50	2.871%	L1 $h_{general}$
qr-decomposition	2.526%	1	25	**34.960%**	L2 h_{hand} 0.001000
sauvola	1.060%	1	50	1.214%	L2 $h_{general}$ 0.000500
chameneos-redux-evtchn	1.002%	1	10	**2.480%**	L2 H_* 0.001000
valet-async	1.852%	15	50	2.597%	$h_{general}$ 0.000050
lu-decomposition	0.000%	-	-	0.611%	L2 $h_{general}$ 0.000500
sieve	0.084%	1	10	**17.397%**	h_{hand} 0.000500
format-benchmark	5.060%	15	25	**6.902%**	L2 H_* 0.000050
chameneos-redux-th	3.509%	5	10	**6.902%**	L2 H_* 0.000500
boyer	**8.639%**	5	10	7.074%	L2 H_* 0.005000

Table 8.1: Results of compilining different variants of the same program using differing command line flags.

In 14 out of the 18 test cases, the Uni-Model Policy manage to find an improvement which is greater than 2%, 4 more than the grid search did.

8.2.3 Is it Random?

Are the speedup obtained in the truely the work of machine learning, or are we simply being fooled by randomness (Taleb 2005)?

In figure 8.3 it is shown in the blue and green plots, where the performance of the benchmark is used to optimised the parameters[2] that only around half the number of models are required to minimize both the mean and max regret. This begs the question - can we select a best model from the set of inlining policies? A box plot of the regret and speedup across a single set of benchmarks is shown in figure 8.6 and 8.7. By eyeballing the two graphs, we can observe that there is a weak linear correlation between median regret and median speedup (linear regression yields an r^2 of approximately 0.55. If we are solely comparing using the median speedup of the dataset, then there is a clear winner. However, there are 2 problems: (1) Median speedup across different benchmarks is ambigious to compare. Flambda does a really good job in inlining CPS code, but does not perform so well with floating point. Our inlining policies are able to exploit the

[2]This is wrong in practice, as this "leaks" information about the evaluation benchmark to the parameter selection procedure.

its lacksture in inlining floating point. For that reason, these two numbers are just not comparable. One belong to the set of benchmarks where Flambda does a decent job, and the other belongs to one where Flambda does not perform as well. It makes sense if we can pre-partition these set of benchmarks into two different groups, but that requires an oracle that recognises where Flambda can improve its inlining policy, and that is exactly what we are trying to do. This ends up as a chicken and egg problem.

The more severe problem (2), however, is shown in figure 8.9. There is non-trivial amounts of correlation between the median speedup and interquartile speedup. This implies that the model with a good "median" is peforming better simply because it has a higher uncertainty in performance the speedup it attains. The regret is a much better metric for measuring the performance of an inlining policy, as its interquartile range (used as a proxy for uncertainty) is pretty much uncorrelated to the regret itself, as shown in figure 8.8. Note that conceptually, it is entirely possible that the median speedup, median regret and the interquartile range stays the same when the interquartile range of the reconstruction error decreases. The benchmarks that contributes to the interquartile range of the speedup need not be the same as those that contributes to those of the regret. For example, if there is a benchmark where $(s_{min}, s_{max} = (4\%, 20\%)$ and the model has a median speedup of 3%. Now if $s_h = 5.5\%$, it will belong in belong in the bottom percentile of the regret, but the top percentile of median. Using the regret attempts to compare, but by introducing additional models (like our inlining policies), we are introducing a fundamental shift to the distribution of execution times. Using the speedup over an existing benchmarks is a mistake akin comparing two measurements with different units. (this once again paints the point that comparing median speedup is not suitable for comparing the inlining policies, and for that matter, programs with completely differnet performance characteristics and distributions).

Selecting the best model solely based on the fact that it has the smallest regret is a bad idea. As shown in figure 8.8, most of the hyperparameter models have an extremely large interquartile range for its regret value. Even when we compared the model with the lowest median regret and the one with the ten-th median regret, there is more than 60% overlap in their interquartile range. We are now back to using the top K models such that J_{mean} and J_{max} are below predefined performance and tolerance thresholds.

Ceteris peribus, median speedup, median regret and the interquartile range of speedup are pretty much tangled up together. Increasing speedup decrease regret, but as a result, all things equal, the interquartile range of speedup increases, ie: the uncertainty of the inlining policy increases in terms of speedup. Without radically reformulation, the best bet of improving the inlining policy is to decrease the regret's interquartile range, hence increasing the certainty of the inlining policy with in terms of the stability of its ability to reconstruct the best known optimal point. A good inlining policy is one that has a low median regret, as well as a low interquartile range of regret. *This is a very alarming sign* - how likely is it that our inlining policies are really just attaining good results by behaving randomly?

Correlating Model Accuracies and Regret

An argument on whether that the the models did indeed manage to learn how to inline fuctions. With ridge-regression-based rewards, the classification accuracy.

Table 8.2 lists the class distribution and prediction accuracies of the inlining policy's models. While the classification accuracies are very high, the classifier did manage to learn how to classify. Relative to baslines of around 48%, the classifier sometimes attains an accuracy of around 63%. The fact that the model *can perform better than a bias guess* suggests that there are statistical regularities between mapping function call sites to inlining decisions.

Using the fields from table 8.2, we tried to correlate the columns of the table and the median regret, with a bit of feature engineering. The resultant plots is shown in figure 8.5, where individual variables are studied for univariate correlation with the median regret. Note prpoortion of apply labels are left out, as it is very much a redundant label. By combining all the variables to instead form a multivariate model, we get an r^2 value of 0.622, as opposed to 0.42 with univariate linear models.

Prportion of Inline	Proportion of Apply	Proportion of I Don't Know	Accuracy	Median Regret
0.044	0.037	0.920	0.937	0.831
0.021	0.018	0.961	0.978	0.778
0.087	0.066	0.847	0.880	0.723
0.014	0.005	0.981	0.987	0.643
0.066	0.039	0.894	0.924	0.623
0.089	0.071	0.840	0.888	0.607
0.251	0.222	0.528	0.651	0.562
0.487	0.404	0.109	0.628	0.555
0.360	0.302	0.338	0.601	0.552
0.383	0.342	0.276	0.596	0.551
0.112	0.069	0.819	0.855	0.551
0.135	0.114	0.750	0.818	0.527
0.293	0.249	0.457	0.648	0.513
0.135	0.109	0.756	0.825	0.473
0.491	0.449	0.060	0.639	0.454
0.272	0.235	0.493	0.686	0.428
0.325	0.281	0.395	0.646	0.421
0.422	0.382	0.196	0.606	0.420
0.180	0.161	0.660	0.743	0.419
0.422	0.377	0.202	0.603	0.389
0.461	0.386	0.153	0.600	0.387
0.478	0.437	0.085	0.635	0.384
0.062	0.048	0.890	0.933	0.362
0.449	0.407	0.143	0.628	0.265

Table 8.2: Table displaying the class distribution, accuracy and median regret of the models.

Figure 8.5 displays the univariate correlation between various features of the classification dataset and the median regret. Without a doubt, univariate predictors make no sense to interpret (how could a higher model accuracy implying higher median regret?). The composite features themselves in the figure is hard to interpret - it is not clera what does the product of two proportions in a class distribution even mean. The $r^2 = 0.622$ when all the features in figure 8.5 are combined to a form a single multivariate linear model. The existence of such a correlation with fairly predictive features (the propotions of of demonstration it has seen, and how well it learns it) suggests the models did not just find a reasonable just by chance.

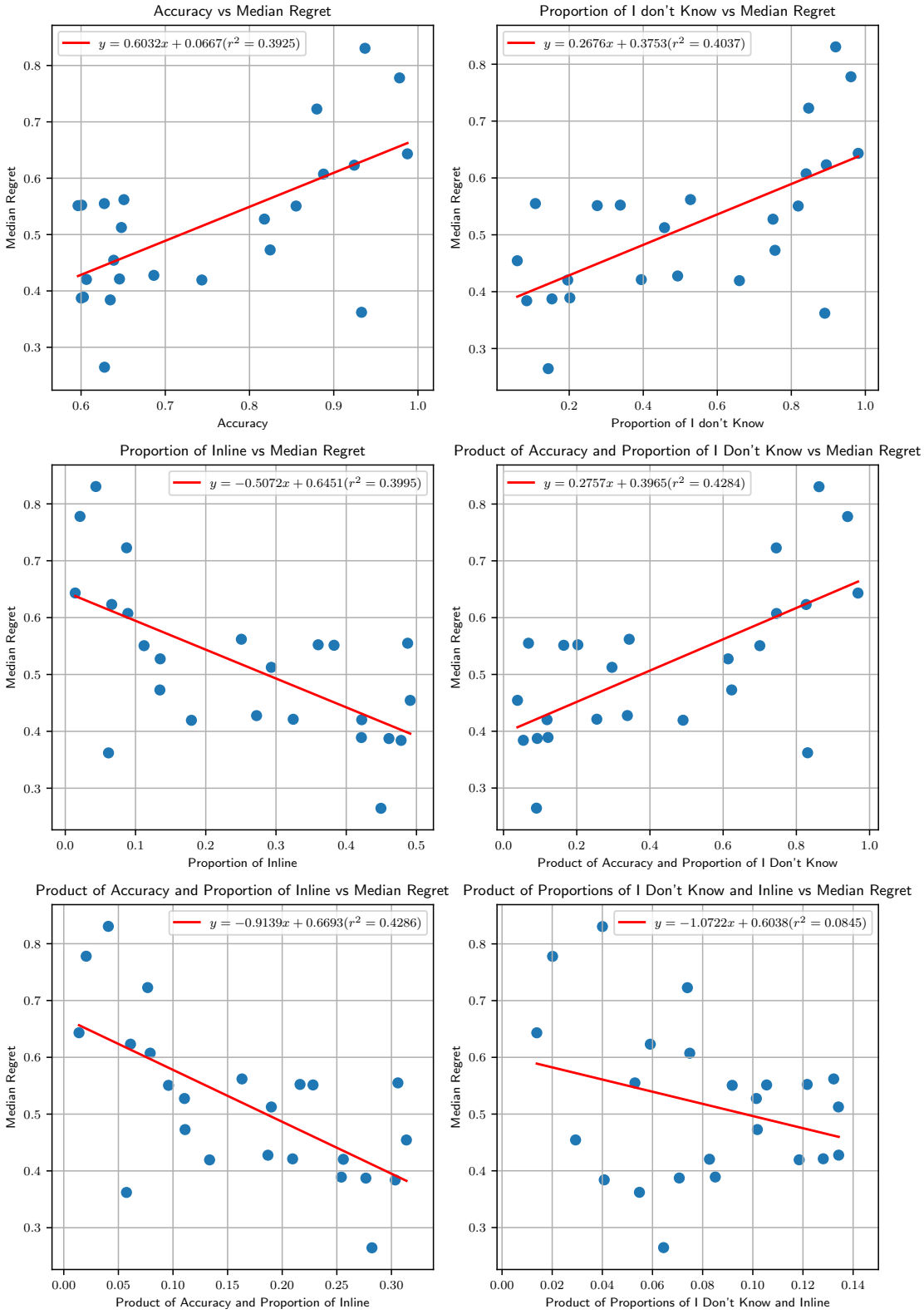


Figure 8.5: Univariate relationship between various features of the training data set with. Combining all these features to form a multi variate linear model yields an r^2 value of 0.622.

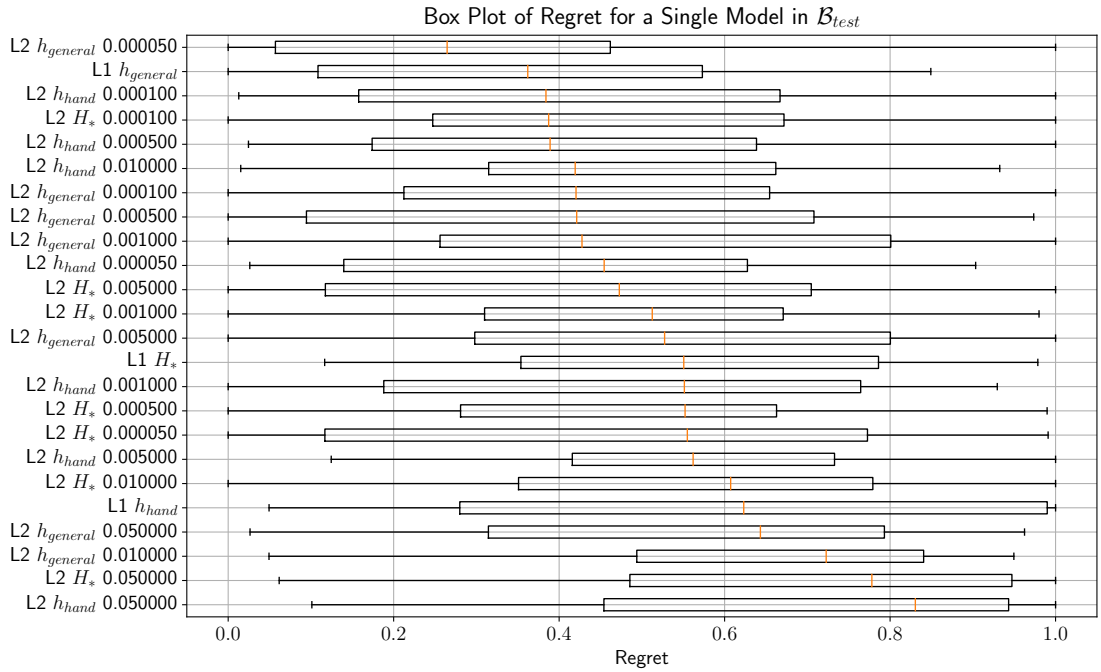


Figure 8.6: Box plot of regret of model across test benchmarks.

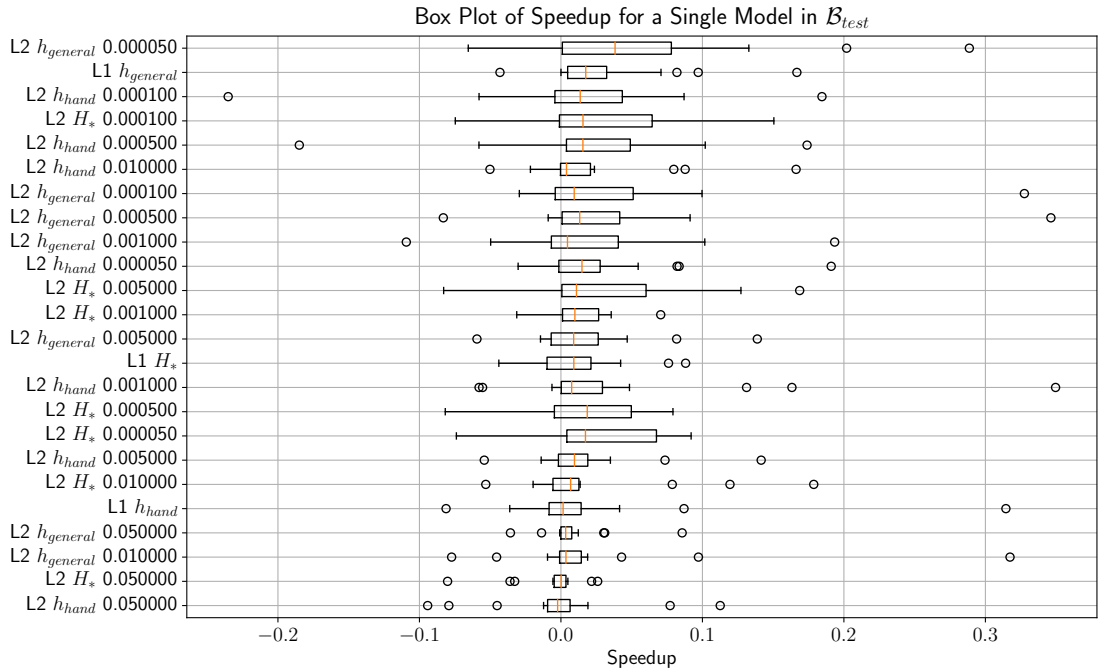


Figure 8.7: Box plot of speedup of model across test benchmarks

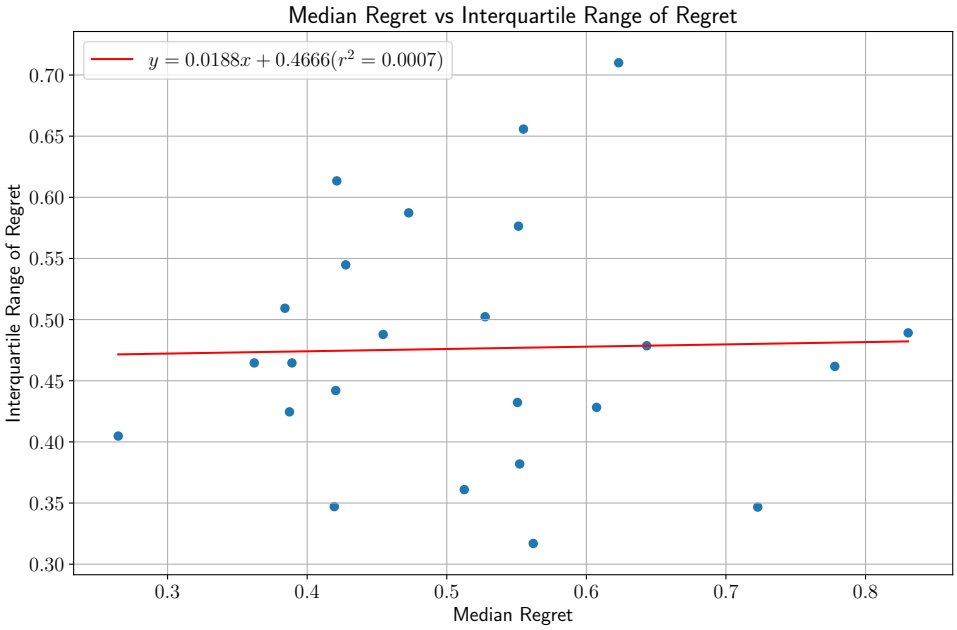


Figure 8.8: Median regret and inter-quartile regert in the test set is uncorrelated.

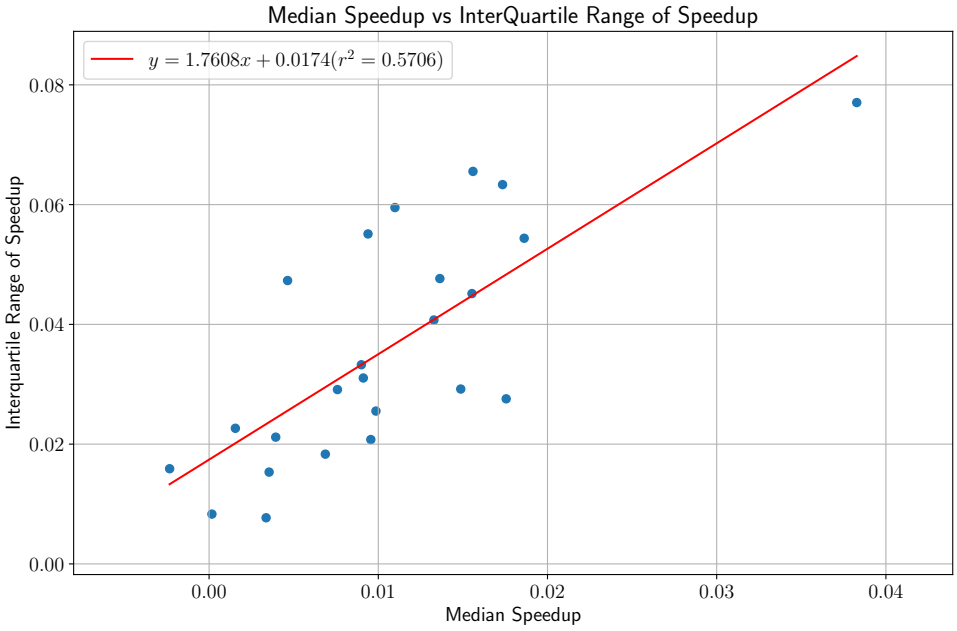


Figure 8.9: Median speedup and inter-quartile speedup exhibits some positive correlation.

Potential Overfitting on the Training Set

As shown in figure 8.2, the inlining policies in general, managed to learn how to speed the program up. In particular we can inspect one of the inlining policies that performs really well in the inlining policies. We compare if the inlining policy captured the general direction suggested by policy modelling. The general direction is defined as (kahan-sum is left out, as a substantial speedup can be obtained by inlining floating point functions that was not exposed during iterative compilation)

Benchmark	Inlining Plicy Speedup	Predictive Modelling Speedup	Captured
almabench	0.873%	0.771%	Yes
lexifi	2.776%	5.201%	Yes
sequence	2.611%	0.170%	No
floats-in-functor	52.853%	55.236%	Yes
fft	2.049%	2.640%	Yes
kahan-sum	5.911%	1.415%	Unclear
bdd	0.239%	3.731%	No
quicksort	-0.083%	1.245%	No
sequence-cps	-0.022%	0.290%	Yes
lens	53.644%	54.203%	Yes
kb	7.352%	10.061%	Yes
hamming	2.321%	3.526%	Yes

8 out of the 11 benchmark suggests that the model managed to pick up the general improvement direction that the predictive model was going for. As we can see. While overfitting is, undesriable in the context of machine learning, it is also suggestive that there is a causation effect in the training set that can be captured. If the training set is a representative sample of the real population (this is an assumption), then being able to model (despite overfitting) implies that the task on hand has a causal relationship that can be modelled.

8.3 CMoE Inlining Policy

To evaluate the CMoE models, we trained to CMoE models: one using the 3 reward assignments obtained via lasso regression (CMoE L1) and one using the 21 reward assignments obtained via ridge regression (CMoE L2). We then benchmarked the two models using the test benchmarks.

It turns out that the performance of CMoE L2 is significantly better than CMoE L1. Two weak hypothesis can be drawn made from this observation:

- CMoE works well only when there is sufficient number of experts. This is a valid assumption due to law of large numbers.
- L2 reward assignment allocates more small local rewards and values throughout the tree, albeit they are noisy. CMoE overcomes this problem, once again, by law of large numbers. In L1 reward assignment, most reward values are sparse, implying that the decisions taken by CMoE and regular L1 will not be very different.

Both of the mentioned hypothesis will require additional data and experiments to prove. Namely, training additional L2 variant of CMoE with different number of observations to study the relationship between number of exprts and the performance of CMoE. This was not investigated in this project due to time constraints[3]. In the text that follows, we will evaluate primarily the L2 CMoE inlining policy. There will be no results section for this policy. The reference comparison for the L2 CMoE inlining policy is the baseline defined earlier. Compared to the baseline Flambda inlining heuristic in the test set, it runs faster ($> 1\%$) in 12 of them, performs worse in 2 of them and performs similarly in 4 of it. This result is shown appendix E.2.

[3]CMoE was introduced 24 hours before this thesis was due

8.3.1 Comparison Against Uni-Model Policy

Based on how CMoE is constructed, we expect the CMoE's performance to be centered roughly around the median. This is because it makes its decisions based on the probability distribution of actions by the other inlining policies. Hence, in general, for every benchmark, if majority of the Uni-Model policies perform well, we expect the resultant policy to perform as well. Figures 8.10 and 8.11 visualises this comparison.

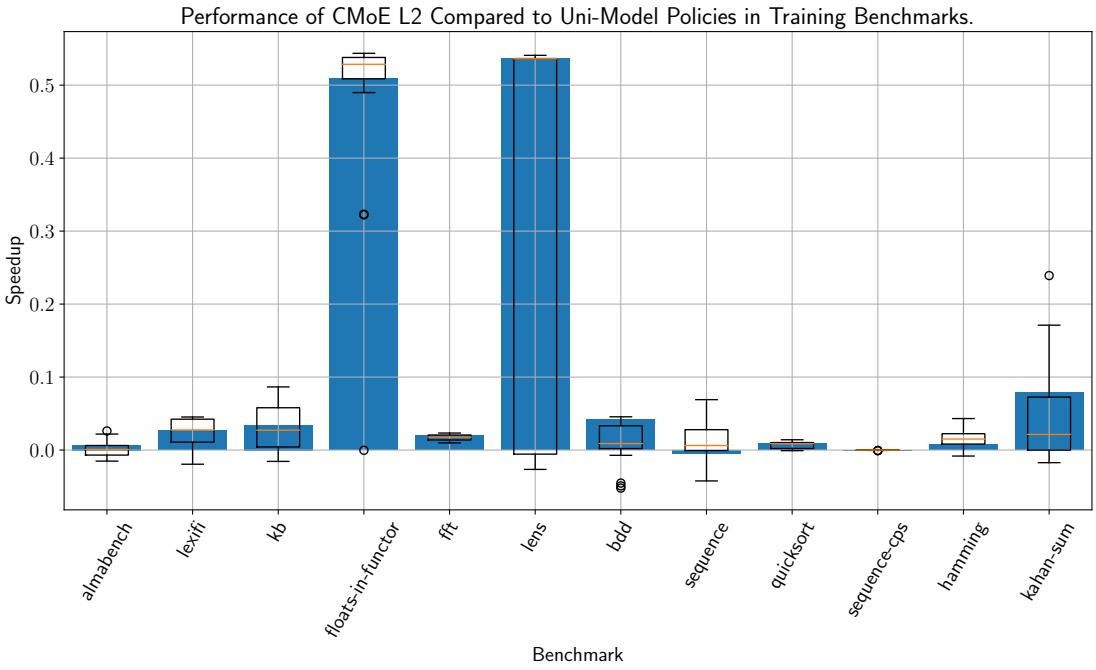


Figure 8.10: CMoE L2 performance relative to other models across the training benchmarks

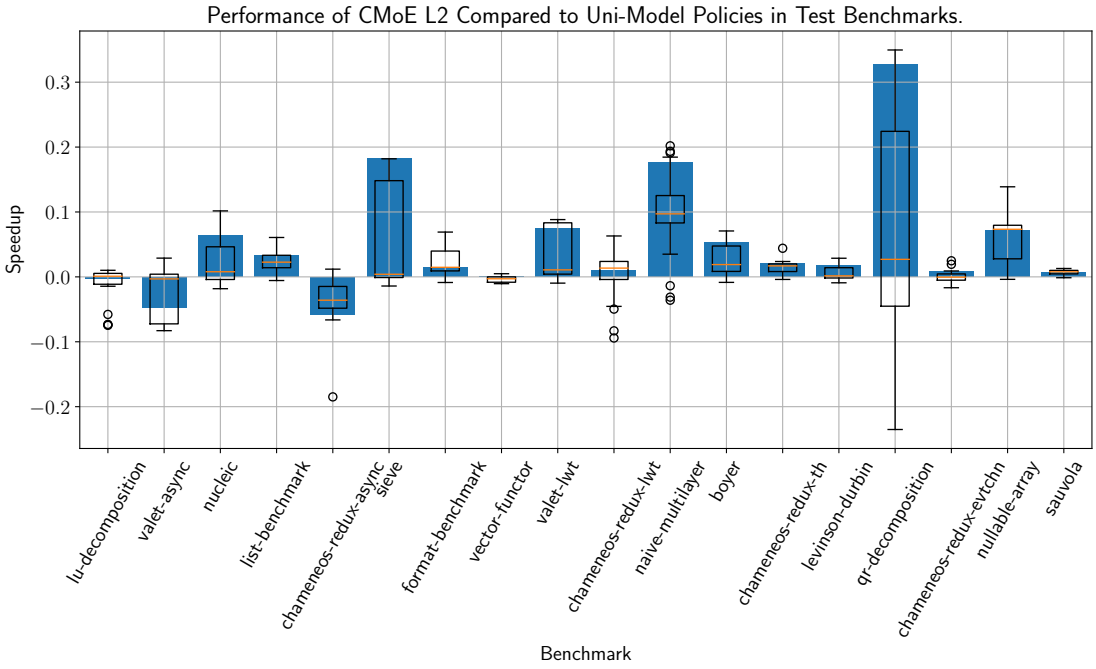


Figure 8.11: CMoE L2 performance relative to other models across the training benchmarks

Figure 8.12 and 8.13 once again shows the box plots for the regret and speedup, with results from L1 and L2 CMoE added. L2 CMoE achieves the lowest interquartile range and range and achieves the second-highest median reconstruction. L2 CMoE achieves the lowest interquartile range and range and achieves the second-highest median reconstruction. This implies that it has learnt a reasonably good policy that has a much lower uncertainty. To roughly visualize what CMoE has learnt, we visualise the speedup attained by CMoE with respect to the other benchmarks. The visualisation is added in figure 8.11 and a special case comparing CMoE L2 with the best uni-kernel model.

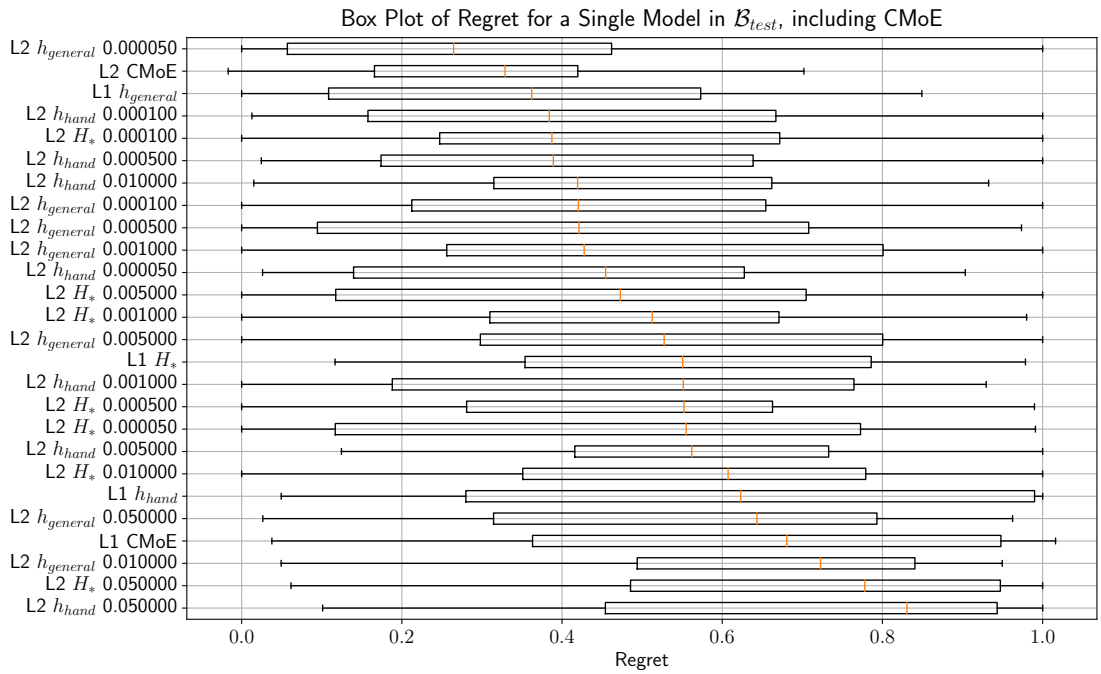


Figure 8.12: Box plot of the regret values in the test benchmarks. CMoE L2 and CMoE L1 plots are added into this plot

From the perspective of overfitting, we can conceptually rely on CMoE L2 policy with more confidence compared to the best uni-model policy (L2 $h_{general}$ 0.00005) on the test set. This is becasue the CMoE L2 policy was constructed without using the benchmarks in the test set to make any decisions. In essence, the test set is used only for benchmarking. The best uni-model policy is selected by choosing the one with the lowest median regret on the test set. If we were to perform a selection without leaking information from the test set, we would have selected a differnt policy, namely L2 H_* 0.00005, which has both a high median regret and interquartile range in regret in the test set. CMOE L2 performs decently (it came 5th) in the training set. It is unlikely thet CMOE L2 ends up being the best model in the training set, as it eliminated differing decisions between different uni-model policy as noise, while those models might have relied on the said decisions get additional speedup (possibly by overfitting) in generating optimal decisions for predictive modelling.

8.3.2 A Pathological Test Case

The two cases in which CMoE L2 performing poorly have some similarities - they both rely on the Async monadic concurrent library (LLC & Contributors 2018*a*). They are `chameneos-redux-async` and `valet-async` and experiments - they both rely on the bind operator, `»=` (a fundamental component for monadic computation (Kock 1970)) with function arguments. That is, they take a high-order annonyous functions as one of its two arguments. Secondly, both of them rely on >>=, most likely with form of code that is not visible in our training example. To verify this, a test case that makes use of annonymous function passed in directly as an argument to the | > operator is used. This benchmark is the `fyq-rev-list` benchmark, which is essentially just 100

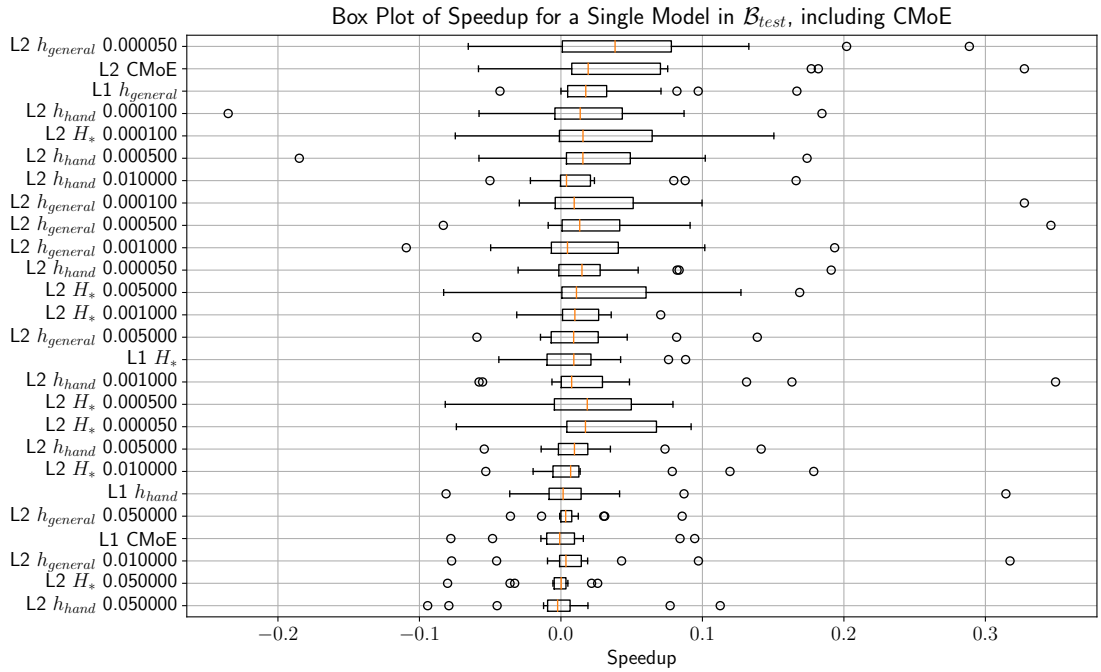


Figure 8.13: Box plot of the speedup in the test benchmarks. CMoE L2 and CMoE L1 plots are added into this plot

annonymous functions chained together. This is a pathological unrealistic test case (who writes 100 annonymous functions chained together using the pipelin operator anyway?), but captures the style of code that our code was struggling with. This pathological test case resulted in a 700% slow down. On hindsight, that sounds like a bad thing, but this actually provides further evidence that the model is picking statistical regularities, given that it misclassifies different test cases with similar properties.

8.4 Summary

In this section, we have evaluated the performance of two compilation pass by the *Uni-Model* and the compilation pass using *CMoE* inlining policies. We introduced the *regret* metric to denote how far away the performance of a model on a benchmark is from known optimality.

The first compilation pass is used to compile multiple versions of the same binary, and the user has to benchmark the binaries to select the fastest version. The second compilation pass, created by the *CMoE inlining policy*, has a much lower range and interquartile range of regret throughout the testing benchmark, denoting that it has a narrower confidence intervals and lower median performance speedup. For that reason, the *CMoE Inlining Policy* is more suited for static compilation, whereas *Uni-Model Policy* is more useful more speculative compilation.

Chapter 9

Conclusion

9.1 Summary

In this thesis, we have presented several ideas to study how machine learning can be used to learn data-guided inlining policies in the FLambda IR for OCaml. A summary of the primary contributions of the thesis is as follows:

- an **Inlining Framework** was proposed (chapter 4). The inlining framework contains a formally proven stable labelling algorithm for tree-based IRs and AST. The algorithm fulfils several properties mutation unrolling (4.1.1), stability (4.1.2) and local uniqueness (4.1.3). On top of the labelling algorithm, the inlining tree data structure was introduced as a means of representing inlining decisions whilst retaining the nested relationship between nested function call sites. Using these two components of the framework, we have implemented a mechanism to override inlining decision sets in the compiler and to dump inlining decision sets on compiling an OCaml module.
- **Iterative compilation** (chapter 5) is used to explore the space of inlining trees for a given set of benchmarks. Every compilation is given a In each of the compilation, a set of pairs, where each pair is comprised of an inlining tree and the corresponding execution statistics, is produced. Two systematic means of sampling were presented, one based on simulated annealing and the other based on a random walk over the space of inlining trees. The former produces highly biased data around lower execution times, whereas the latter produces a more uniform sample across the space of execution times (and inlining trees).
- The next stage of the pipeline, **Call Site Reward Assignment** (chapter 6) attempts to assign a numerical value to function call sites for inlining decisions. Prior to any modelling, all the inlining trees are transformed into expanded trees. Expanded trees are an alternative representation of inlining trees, with the main difference being that their set of children are invariant to any inlining transformations. In that chapter, the incremental tree-space exploration problem has also been proposed. It defines the problem where we try to find the best tree from the set of possible trees for the problem definition that has the highest total value V^*. Using linear regression with various means of regularisation, we can fit use past demonstrations to leaf inline and apply rewards for nodes in the said tree. As a corollary of this, using predictive modelling techniques, we can generate an "optimal" set of inlining decisions. The main goal of this chapter is that all function call sites have a reward assignment, namely, they have the values $\mathcal{R}_{inline}$, $\mathcal{R}_{apply}$ and V^*
- **An inlining policy** (chapter 7) is then trained by deriving classification labels from data generated from reward assignment. As we have multiple reward assignments due to choice of hyperparameters, we have multiple inlining policies. This results in two forms of inlining policies, one which is more speculative uses each of the a reward assignment to train a single policy and the other, which utilises multiple reward assignments to train a policy. The latter combines together using the Chaotic Mixture of Experts (CMoE) model to smooth out unfamiliar data points and make more conservative inlining decisions. The former policy

is suited for scenarios in which the user compile multiple copies of a single binary and selects via benchmarking, whereas the latter is more suited for a more standard static one-off compilation.

9.2 Future Work

Flambda

The inlining framework and stable labelling algorithm described in chapter 4 was a fundamental building block throughout the optimisation pipeline in this project. It is likely that the convenience that can be attained from a labeling framework will be useful for reporting inlining decisions in a programmatically parsable manner. (Flambda allows the user to read in the form of emacs notes, but they are not programatically parsable). This will also enable anyone who wishes to study function inlining in the OCaml to do so without the need to extensively hack the compiler.

The OCaml compiler has an eccentric feature, whereby the user can pass a code plugin that gets dynamically invoked (Leroy et al. 2017). The code has access to modules in the compiler which exposes an interface via a `.mli` file. This feature was used in this compiler to override the default function inlining heuristics, as discussed in chapter 7 which the project's custom machine-learning based inlining policy. The interface for the custom inlining policies in this project is a function to set a ref to reference a function that should be invoked. The policy function has a type signature of `val f : Inline_and_simplify_aux.Env.t -> Inlining_query.t lazy_t -> decision option`. The first argument is an immutable context object that tracks various details whilist walking the Flambda IR, such as mapping of variables to available approximations, function unrolling depth and inlining depth. The inlining query is a data structure introduced to wrap several possible source of features, such as the Flambda expression of the function declaration's body and the globally unique identifier (introduced in chapter 4) of the function call site.

An other thing that we have noticed is that the inlining heuristic *and* predictive modelling *and* simulated-annealing function inlining tends to inline recursive function calls within the function declaration itself, something Flambda, as of time of writing, does not do[1].

Sharing the Work

Some work should be put into sharing the code used to produce the findings presented in this thesis.

9.2.1 Thorough Benchmarking

The benchmarks in used in this project are primary microbenchmarks. There is larger suit of programs that are readily availble for benchmarking in *ocamlbench repo*[2]. The experimental framework, is currently only suited only microbenchmarks where it can be readily compiled by invoking `make`.

One of the problems discussed when evaluating the compilation pass in section 8.2.3, is that there is not very strong evidence that the benefits obtained from inlining are not simply a result of randomness. Despite the arguments provided in beneath that section, nothing beats proof from evidence in evaluating a large number of benchmarks. [3] Benchmarking the inlining pass' performance on real-world programs (rather than designed benchmarks) will give us much better confidence.

[1] Line 537 of https://github.com/ocaml/ocaml/blob/b5d1929e878d6d2ccfe6ea349960ee803e9e45c6/middle_end/inlining_decision.ml

[2] https://github.com/OCamlPro/ocamlbench-repo/tree/master/packages

[3] Actually, by running a lot of benchmarks, we are simply reinforcing the conjecture that the inlining policy has made use of machine learning effectively.

9.2.2 Enhancing the Optimisation Pipeline

Generating More Diverse Data

The model is trained on inlining decisions based on 12 benchmarks. 12 benchmarks is hardly representative of the kinds of programs that can be obtained in the real world. Furthermore, the 12 benchmarks used in this project are completely self-contained, that is, they depend only on the OCaml standard library. A step forward from performing exploration solely using self-contained microbenchmarks is to perform training using microbenchmarks that depends on libraries from the `OPAM` package manager. Through importing packages from a larger range of codebases, the model will be exposed to a more diverse and representative set of function declarations call sites. This will enable the model to generalise better to real-world programs. The challenge in iterative compilation on larger program is that it will take a much higher number of itereations for iterative compilation to even find something remotely useful. This is especially hard on programs with sparse rewards.

Feature Engineering

The set of features chosen in appendix B were not selected with thorough investigation. As shown in figure 7.2 and discussed in chapter 7, there is noticable correlation between features vectors in the training data. Hence, a low hanging fruit to improve the optimisation pipeline is to simply engineer better sets of features.

Hyperparameter Selection for Reward Assignment

A more automated hyperparameter selection mechanism will be useful for the reward assignment phase. This is useful as the time taken for running a single benchmark is quite long ($5x10.0s$ per execution of a benchmark). This would be similar to how we have shown in chapter 6 that the λ parameter in L1-regularisation variant of solving the reward vector.

9.2.3 Speculative Research Directions

Static Exploration of for Inlining Opportunities

An important thesis presented in this piece of work is specifying the distinction the difference between the long-term reward between inlining a function call $\mathcal{V}^*$ and the immediate reward $\mathcal{R}_{inline}$. Choosing not to inline the function will instead yield a apply / termination reward, $\mathcal{R}_{apply}$. As presented in the background, Cummins et al. (2017*b*) suggested that the sparsiy of avilable benchmarks makes feature engineering a hard task, whereas Wang & O'Boyle (2018) suggests that the input to training machine learning algorithms for compilers ought to be a obtained from a representative distribution of programs written in user space.

Function inlining in high-level languages are a special case - they are more exploratory, rather than attempting to seek immediate benefit. That is, that is, the primary benefit attained from a inlining function call is less of removing the overhead of the function call, but rather, unveiling further inlining decisions that can result into large substantial benefits (such as those in the `lens` experiment). Assuming the existence of an oracle $\mathcal{R}$ function, the problem will be reduced to creating a model of $\mathcal{V}^*$ based on the orcale function and utilise static exploration on a representative distribution of OCaml programs. This problem formulation bears similarity to Q-Learning in machine learning that is used to solve markov decision processes.

MCTS-based Dynamic Exploration

The solution presented in 6 uses offline fitting to perform reward assignment based on past execution of the program with different sets of inlining decisions as demonstration. An alternative method

of reward assignment is to perform dynamic exploration, learning $\mathcal{R}$ on the go while exploring the expanded tree.

This solution attempts to solve $\mathcal{R}$ a single benchmark for a set of representative input data. This form of solution exploits, for a given u, the linear independence in $\{\mathcal{R}(v)|v \in \mathcal{C}(u)\}$. As compared to the offline method, the localised reward assigned to every node is rather "How much better is inlining a function compared to branching to it as run-time". As a consequence, this model of reward assignment assumes $\forall v \in \mathcal{V}.\mathcal{R}(v, Apply) = 0$ and attempts to model only $\forall v \in \mathcal{V}.\mathcal{R}(v, Inline)$, effectively reducing the number of parameters of the problem.

The key idea of the algorithm is to incrementally build the tree and update the reward function based on a minimal tree structure. Algorithm 4 performs the said gradual dynamic exploration solving $\mathcal{R}_{inline}(u)$ at every step of the exploration. The leaf node that is chosen as the next candidate for exploration is sampled based on the perceived value and the code coverage.

Input: The parameters defining an instance of the node-reward assignment problem
Output: $\mathcal{V} \to \mathbb{R}$, the learnt $\mathcal{R}_{inline}$ for the given program
Function `Explore`$(root, \mathcal{V}, \mathcal{C}, \mathcal{D}, \gamma, f_{benefit})$**:**

```
leaves ← {root}
value_prev ← 0
decisions ← {}
frequency ← {}
decisions[root] ← Apply
foreach u ∈ V do
    frequency[u] ← 0
end

for N := 0..N_max do
    u ← Select(root, C, leaves, N, frequency)
    decisions(u) ← Inline
    for v ∈ C(u) do
        decisions(v) ← Apply
    end

    (exec_times, coverage) ← exec(decisions, D)
    for (v, c) ∈ coverage do
        frequency[v] ← frequency[v] + c
    end

    value ← f_benefit(exec_times)
    R_inline(u) ← γ^(−depth(u)) × (value − value_prev)
    value_prev ← value
end

return R_inline
```

Algorithm 4: Dynamic Inlining Tree Exploration

The *exec* procedure in the above algorithm executes the benchmark across all the input data in $\mathcal{D}$ and returns a tuple containing the list of execution times and a node-coverage mapping. The coverage mapping conforms to the mathematical property that $\forall v \in \mathcal{V}.coverage_v \in [0, 1]$, and hence can be viewed as the probability a node is explored across the input dataset. $f_{preprocess}$ defined similar as those in chapter 6. Unlike $f_{preprocess}$ that computes the transformation of the value relative to some notion of reference, $f_{preproecss}$ computes the value of the tree relative to the instance where all nodes are not inlined.

In a markov chain problem formulation, $P(s, s')$ refers to the probability of transitioning from a state s transitions to some other state s'. $P(u, v)$ passed as an argument of $f_{estimate}$ below refers to dominance, rather than a transition probability. It can be seen as measure of how important or dominant a particular edge $u \to v$ is compared to other edges pointing out from u. Code paths

that are taken more often than others will have higher dominance[4]

The *Select* function requires special attention. The purpose of the function is to selectively explore nodes that will yield interesting inlining paths. Determining "interesting paths" statically is hard, but can be easily done via dynamic analysis. Namely, nodes that are not reachable by the given input dataset should not require further exploration. The *Select* function can be defined as:

- Uniformly sample leaf based on the dominance from its parent node defined by $P(u, v)$.
- Uniformly sample the nodes with an additional bias term defined by its parent's inlining benefit.

The goal of performing this dynamic and incremental exploration is to try to find inlining opportunities that are possibly "far" awway from the default inlining heuristic. Starting from a clean-state, rather than a simulated annealing sink of a "good default" can possibly result in more performant code (Schkufza et al. 2013).

[4]This is, to some extend, a bad abuse of notation. But the notation $P(.,.)$ is retained to maintain the interoperability with standard MDP exploration algorithms.

Appendix A

Algorithms for Inlining Trees

The code / pseudocode presented in this section refers to algorithms that are not critical to help understanding thesis, but interesting enough that they warrant presentation in the appendix. OCaml algorithms here assumes that it uses the Core standard library (LLC & Contributors 2018*b*), rather than the standard library.

A.1 Computing Diff between Inlining Trees

The time complexity of the algorithm is $O(N_{nodes} \times E_v[|\mathcal{C}(v)|])$.

```
let shallow_equality a b =
  match a, b with
  | Apply a, Apply b ->
    a.apply_id = b.apply_id
  | Declaration a, Declaration b ->
    a.apply_id = b.apply_id
  | Apply_non_inlined_function a, Apply_non_inlined_function b ->
    a.apply_id = b.apply_id
;;

let diff ~(left : root) ~(right : root) =
  let shallow_diff ~left ~right =
    let shallow_equal = ref [] in
    let left_only = ref [] in
    let right_only = ref [] in
    List.iter left ~f:(fun t ->
      if List.exists right ~f:(shallow_equality t) then
        same := (t, List.find_exn right ~f:(shallow_equality t)) :: !same
      else
        left_only := t :: !left_only);
    List.iter right ~f:(fun t ->
      if not (List.exists left ~f:(shallow_equality t)) then
        right_only := t :: !right_only;
    );
    (Same !shallow_equal, Left_only !left_only, Right_only !right_only)
  in
  let rec loop ~(left : t list) ~(right : t list) =
    let (Same same, Left_only left_only, Right_only right_only) =
      shallow_diff ~left ~right
    in
    let descent =
      List.map same ~f:(fun (left, right) ->
```

```
        let diffs =
          match left, right with
          | Declaration l_decl, Declaration r_decl ->
            loop ~left:l_decl.children ~right:r_decl.children
          | Apply_inlined_function l_inlined, Apply_inlined_function r_inlined ->
            loop ~left:l_inlined.children ~right:r_inlined.children
          | Apply_non_inlined_function _, Apply_non_inlined_function _ ->
            []
          | _, _ -> assert false
        in
        List.map diffs ~f:(fun (diff : Diff.t) ->
          let common_ancestor = diff.common_ancestor in
          { diff with common_ancestor = left :: common_ancestor }))
      |> List.concat_no_order
      |> List.filter ~f:(fun diff ->
          match diff.left, diff.right with
          | [], [] -> false
          | _, _ -> true)
    in
    let current =
      match left, right with
      | [], [] -> None
      | _, _ ->
        Some {
          Diff.
          left  = List.map left_only ~f:(fun x -> Left x);
          right = List.map right_only ~f:(fun x -> Right x);
          common_ancestor = [];
        }
    in
    match current with
    | Some hd -> hd :: descent
    | None -> descent
  in
  loop ~left ~right
;;
```

A.2 Constructing Decision Sets from Inlining Trees

A.2.1 Maximal decision set

```
let to_maximal_decision_set ~round (root_ : t) =
  let open Data_collector in
  let build_decisions ~trace:old_trace ~source ~path ~applied ~action =
    let apply_id = Apply_id.of_path_inconsistent path in
    let acs = { Trace_item. source; applied; apply_id; } in
    let metadata = applied in
    let acs = Trace_item.At_call_site acs in
    let trace = acs :: old_trace in
    let decisions =
      { Decision.
        round; trace = acs :: old_trace;
        apply_id; action; metadata;
      }
    in
    (trace, [decisions])
  in
```

```
  let rec loop ~trace ~previous root =
    List.concat_map root ~f:(function
      | Decl decl ->
        let enter_decl =
          { Trace_item. source = previous; declared = decl.func }
        in
        let enter_decl = Trace_item.Enter_decl enter_decl in
        let previous = Some decl.func in
        let trace = (enter_decl :: trace) in
        loop ~trace ~previous decl.children

      | Inlined inlined ->
        let trace, decisions =
          build_decisions ~source:previous ~action:Action.Inline ~trace
            ~path:inlined.path ~applied:inlined.func
        in
        let previous = Some inlined.func in
        let children = inlined.children in
        decisions @ (loop ~trace ~previous children)

      | Apply apply ->
        let (_trace, decisions) =
          build_decisions ~source:previous ~action:Action.Apply ~trace
            ~path:apply.path ~applied:apply.func
        in
        decisions)
  in
  loop ~trace:[] ~previous:None root_
;;
```

A.3 Constructing Expanded Tree from Inlining Trees

The implementation of this algorithm is massive, declaration of functions whose definitions are obvious from their names have been omitted.

```
let expand_decisions_in_call_site_based_on_inlining_path input_tree =
  let remove_last_node_exn l =
    List.rev (List.tl_exn (List.rev l))
  in
  let expand_inlining_path_parents
      ~child_node
      ~history
      ~inlining_path:this_inlining_path =
    assert (Int.(>) (List.length this_inlining_path) 0);
    (* Recall that the path is from top to bottom, that is if
     *   [f_a] -> [f_b] -> [f_c] -> [f_d]
     * This is /a/b/c/d
     *)
    let path_prefix = history in
    let path_to_patch =
      let trimmed = trim_prefix ~prefix:path_prefix this_inlining_path in
      begin match trimmed with
      | [] ->
        failwithf !"trimed is empty this = %{Apply_id.Path} prefix = %{Apply_id.Path}"
          this_inlining_path path_prefix ()
      | _ -> ()
      end;
```

```
      remove_last_node_exn trimmed
    in
    (* We construct the node bottom-up to prevent stack-overflow
     * (this recursive depth can be potentially very very deep) *)
    let rec patch_bottom_up ~child ~rev_path =
      match rev_path with
      | [] -> child
      | _ :: tl ->
        let inlined =
          { E.
            func     = E.expanded_function_metadata;
            path     = history @ List.rev rev_path;
            children = [ child ];
          }
        in
        let child = E.Inlined inlined in
        patch_bottom_up ~child ~rev_path:tl
    in
    patch_bottom_up ~child:child_node ~rev_path:(List.rev path_to_patch)
  in
  let rec loop ~history root =
    List.map root ~f:(function
        | Declaration decl ->
          let children = loop ~history:[] decl.children in
          E.Decl {
            children = children;
            func     = decl.declared;
          }

        | Apply_inlined_function inlined ->
          let inlining_path = Apply_id.to_path inlined.apply_id in
          let children = loop ~history:inlining_path inlined.children in
          Log.Global.sexp ~level:Debug
            [%message (inlining_path : Apply_id.Path.t) (history: Apply_id.Path.t)];
          let child_node =
            E.Inlined {
              func     = inlined.applied;
              path     = inlining_path;
              children = children;
            }
          in
          expand_inlining_path_parents ~child_node ~history ~inlining_path

        | Apply_non_inlined_function non_inlined ->
          let inlining_path = Apply_id.to_path non_inlined.apply_id in
          let child_node =
            E.Apply {
              func     = non_inlined.applied;
              path     = inlining_path;
            }
          in
          expand_inlining_path_parents ~child_node ~history ~inlining_path)
  in
  loop ~history:[] input_tree
;;

let rec merge_decisions_in_call_site (tree : Expanded.t) =
  let used = Array.create ~len:(List.length tree) false in
```

```
  let tree = Array.of_list tree in

  Array.mapi tree ~f:(fun i this_node ->
      if used.(i) then
        None
      else begin
        let similar_siblings =
          let ret = ref [] in
          for j = i + 1 to Array.length tree - 1 do
            if not used.(j) then begin
              let another_node = tree.(j) in
              if
                Expanded.node_equal_without_descend another_node this_node
              then begin
                used.(j) <- true;
                ret := another_node :: !ret
              end
            end
          done;
          List.rev !ret
        in
        let additional_children =
          List.concat_map similar_siblings ~f:(function
              | E.Inlined { children; _ } -> children
              | E.Decl { children; _ } -> children
              | E.Apply _ -> [])
        in
        used.(i) <- true;
        let func =
          this_node :: similar_siblings
          |> List.filter ~f:(fun node ->
              not (Function_metadata.equal
                E.expanded_function_metadata (E.get_func node)))
          |> List.hd
          |> Option.map ~f:E.get_func
          |> Option.value ~default:(E.get_func this_node)
        in
        let this_node = E.update_func this_node func in
        match additional_children with
        | [] -> Some this_node
        | additional_children ->
          Some (
            E.update_children_exn this_node (
              E.get_children_exn this_node @ additional_children))
      end)
  |> Array.to_list
  |> List.filter_opt
  |> List.map ~f:(function
      | E.Inlined inlined ->
        let children = merge_decisions_in_call_site inlined.children in
        E.Inlined { inlined with children }
      | E.Decl decl ->
        let children = merge_decisions_in_call_site decl.children in
        E.Decl { decl with children }
      | E.Apply a -> E.Apply a)
;;

let fill_in_decisions_from_declaration input_root =
```

```
let declaration_map = ref Closure_id.Map.empty in
let replay_declaration_inlined_leaves inlined_root
    ~inlining_path ~declaration_root =
  let open Option.Let_syntax in
  let%map () =
    let cond =
      List.for_all inlined_root ~f:(fun node ->
          E.is_decl node ||  (* declarations are noisy, due to
                                function specialisation *)
          List.exists declaration_root ~f:(fun decl_node ->
              let node =
                E.update_path_exn node
                  (trim_prefix ~prefix:inlining_path (E.get_path_exn node))
              in
              (* Important that the tags match here. Cannot have Inlined
               * in one and not inlined in another. (Recursive functions
               * that inline the body).
               *)
              Option.equal Apply_id.Path.equal
                (E.get_path decl_node)
                (E.get_path node)))
    in
    if cond then Some () else None
  in
  let nodes_to_replay =
    E.filter_map declaration_root ~f:(fun node ->
        match node with
        | E.Decl    _ -> None
        | E.Inlined inlined ->
          let path = inlining_path @ inlined.path in
          Some (E.Inlined { inlined with path })
        | E.Apply _ ->  None)
    |> List.rev
  in
  let replayed_paths =
    let ret = ref Apply_id.Path.Set.empty in
    E.iter nodes_to_replay ~f:(function
        | E.Inlined { path; _ } -> ret := Apply_id.Path.Set.add !ret path
        | _ -> ());
    !ret
  in
  replayed_paths, List.fold nodes_to_replay ~init:inlined_root ~f:(fun root node ->
      E.add_node_to_parent_head root node)
in
let rec loop ~replayed root =
  List.map root ~f:(function
      | E.Decl decl ->
        let children = loop ~replayed:Apply_id.Path.Set.empty decl.children in
        let decl = { E. children = children; func = decl.func } in
        let closure_id =
          let message =
            Format.asprintf "%a" Closure_origin.print
              decl.func.closure_origin
          in
          Option.value_exn ~message decl.func.closure_id
        in
        Log.Global.sexp ~level:Debug [%message (closure_id : Shadow_fyp_compiler_lib.Clo
        declaration_map := (
```

```
              Closure_id.Map.add closure_id decl !declaration_map);
            E.Decl decl

        | E.Inlined inlined ->
          let open Option.Let_syntax in
          let inlining_path = inlined.path in
          if Int.(>) (List.length inlining_path) 100 then begin
            failwithf !"inlining path too long: %{Apply_id.Path}"  inlining_path ()
          end;
          Log.Global.sexp ~level:Debug [%message (inlining_path : Apply_id.Path.t)];
          let new_replayed, new_children =
            let opt =
              let%bind closure_id = inlined.func.closure_id in
              let%bind decl =
                Closure_id.Map.find_opt closure_id !declaration_map
              in
              if Apply_id.Path.Set.mem replayed inlining_path then
                None
              else
                replay_declaration_inlined_leaves inlined.children
                  ~inlining_path:inlined.path
                  ~declaration_root:decl.children
            in
            match opt with
            | None -> (Apply_id.Path.Set.empty, inlined.children)
            | Some (a, b) -> (a, b)
          in
          let replayed = Apply_id.Path.Set.union new_replayed replayed in
          let new_children = loop ~replayed new_children in
          E.Inlined { inlined with children = new_children }

        | E.Apply apply -> E.Apply apply)
  in
  loop ~replayed:Apply_id.Path.Set.empty input_root
;;

let rec remove_bogos_apply_nodes root =
  List.filter_map root ~f:(fun node ->
      match node with
      | E.Inlined inlined ->
        let children = remove_bogos_apply_nodes inlined.children in
        Some (E.Inlined { inlined with children })
      | E.Decl decl ->
        let children = remove_bogos_apply_nodes decl.children in
        Some (E.Decl { decl with children })
      | E.Apply apply ->
        let matches_apply_id = function
          | E.Inlined inlined ->
            Apply_id.Path.equal inlined.path apply.path
          | _ -> false
        in
        if List.exists root ~f:matches_apply_id
        then None
        else Some node)
;;

let rec remove_stubs ~new_prefix tree =
  let is_a_single_stub_apply children =
```

```
    match children with
    | [ E.Apply apply ] ->
      let last_component = List.hd_exn (List.rev apply.path) in
      begin match snd last_component with
      | Apply_id.Stub -> true
      | _ -> false
      end
    | _ -> false
  in
  List.concat_map tree ~f:(fun node ->
      match node with
      | E.Decl decl ->
        [ E.Decl { decl with children = remove_stubs ~new_prefix:[] decl.children } ]
      | E.Inlined inlined ->
        let last_component = List.hd_exn (List.rev inlined.path) in
        let new_path =
          match snd last_component with
          | Apply_id.Stub -> new_prefix
          | _ -> new_prefix @ [last_component]
        in
        let children =
          let new_prefix = new_path in
          remove_stubs ~new_prefix inlined.children
        in
        begin match snd last_component with
        | Apply_id.Stub -> children
        | _ ->
          let path = new_path in
          (* We use the old children (before removing stub) because any
           * lone stub would have been an empty list now.
           *)
          if is_a_single_stub_apply inlined.children then
            [ E.Apply { func = inlined.func; path; } ]
          else
            [ E.Inlined { inlined with children; path; } ]
        end
      | E.Apply apply ->
        let last_component = List.hd_exn (List.rev apply.path) in
        begin match snd last_component with
        | Apply_id.Stub -> []
        | _ -> [ E.Apply { apply with path = new_prefix @ [last_component] } ]
        end
    )
;;

let expand root =
  root
  |> expand_decisions_in_call_site_based_on_inlining_path
  |> merge_decisions_in_call_site
  |> fill_in_decisions_from_declaration
  |> merge_decisions_in_call_site
  |> remove_bogos_apply_nodes
  |> remove_stubs ~new_prefix:[]
;;
```

Appendix B

Features

These are the features used for feature engineering. Symbols and variables are different ways values can be referenced. A value approximation is the statically known set of possible values that a variable / symbol can exhibit.

feature	description
Callee Body Features	
Symbol Approximations	Number of symbols that have a value approximation.
Variable Approximations	Number of variables that have a value approximation.
Move with Set of Closures Approximations	OCaml has an optimisation that allows a function to statically know the offset in bytes from one closure to another in a set of closures. This approximation is the offset. This features counts the number of such approximations.
Project Var Approximations	Number of variable projection approximations. A projection approximation is the return value from reading an entry in a closure block.
Number of Const Int	A constant integer. This compiles into a word in the instruction.
Number of Const Pointer	Number of variants that are known to be a simple integer, rather than a block.
Size	Size of the Flambda IR body.
Declaration Features	
Is a Functor	whether the function is an OCaml functor
Only use of Function	whether this is definitely the only use of function. This will be true for all anonymous functions tha are not bind to assigned to any variables.
Number of Invariant params	Number of parameters that are independent on the function call in recrusive scenarios. These parameters ca be removed by function specialisation
Number of Parameters	The number of parameters, including invariant params.
Argument Features	
Type of Argument 0 to 6	The statically known type of argument. The types can be one of*Block, Int, Char, Constptr, Boxed Int, String, Float Array, Set of Closures, Closure, Extern, Symbol, Unknown, Bottom, Unresolved or Unused.*
Environment Features	
Inlining Level	the amount of function inlining the compiler has performed.
Closure depth	the number of function declarations wrapping a said function call
Branch depth	the number of branches (swithces, conditionals) wrapping the branch
direct call	whether the function call will be compiled to a static branch (as opposed to calling a function via pointer)
Flambda Heuristics	

Remove Call	The number of function calls removed.
Remove Allocation	Number of allocations removed from inlining a function
Remove Branch	Number of branches removed from inlining a function
Remove Prim	Number of primitive operations that are removed from inlining a function
Structural Features	
Apply	These features are the kind of nodes that can eppear in the Flambda IR. The structural features is simply the number of said kind of nodes.
Send	
If then else	
Switch	
String Switch	
static Raise	
Try with	
Whike	
For	
Proved Unreachable	
Symbol	
Const	
Allocated Const	
Read Mutable	
Read Symbol Fields	
Project Closure	
Move within set of closures	
Project Var	
Primitive Ops	

Table B.1: Exhaustive List of Features Used for training

Appendix C

Compilation Script

This compilation script binary searches the largest value for `inline-max-unroll` that can be used to compile a given program.

```
#!/bin/bash

# ocamlopt.opt - a wrapper around ocamlopt.opt to binary search inline-max-unroll
# To use this script to compile arbitrary program, add the script's directory
# to the FRONT of your PATH variable and run ocamlopt.opt / make as usual.

set -euo pipefail

hi="15"
lo="0"

BEST_SO_FAR=""
COMPILE_ARGS="$@"

if [ ! -z ${ORIGINAL_OCAMLPARAM+x} ]; then
  ORIGINAL_OCAMLPARAM="$OCAMLPARAM"
fi

if [ -z ${OCAMLOPT_TIMEOUT+x} ]; then
  TIMEOUT="20s"
else
  TIMEOUT="$OCAMLOPT_TIMEOUT"
fi

try_compiling(){
  local mid=$1
  THIS_OCAMLPARAM="O3=1,inlining-report=0,dflambda=0,inline-max-unroll=$mid"

  echo "[OPTIMISER] Trying $mid ..."
  set +e
  timeout "$TIMEOUT" /home/fyquah/fyp/opam-root/4.06.0+fyp/bin/ocamlopt.opt \
    $COMPILE_ARGS
  EXIT_CODE="$?"
  set -e
}

echo "[OPTIMISER] Compiler timeout = $TIMEOUT"
echo "[OPTIMISER] Speculatively trying $hi"
```

```
try_compiling "$hi"
if [ $EXIT_CODE = "0" ]; then
  exit 0
fi

while [ ! "$lo" -eq "$hi" ]; do
  mid=$((hi + lo))
  mid=$((mid / 2))

  echo "[OPTIMISER] lo=$lo mid=$mid hi=$hi"

  if [ -z ${ORIGINAL_OCAMLPARAM+x} ]; then
    export OCAMLPARAM="_,$THIS_OCAMLPARAM"
  else
    export OCAMLPARAM="$ORIGINAL_OCAMLPARAM,$THIS_OCAMLPARAM"
  fi

  try_compiling $mid

  if [ "$EXIT_CODE" -eq 0 ]; then
    echo "[OPTIMISER]   $mid SUCEEDED!"
    BEST_SO_FAR="$mid"
    lo=$((mid+1))
  else
    echo "[OPTIMISER]   $mid FAILED."
    hi="$mid"
  fi
done

if [ "$BEST_SO_FAR" = "" ]; then
  exit 121
else
  echo "[OPTIMISER] Accepted $BEST_SO_FAR!"
  echo "[OPTIMISER] Accepted $BEST_SO_FAR!" >/dev/stderr
  exit 0
fi
```

Appendix D

Technical Specifications

D.1 Benchmark Compilation

- **OCaml** - a fork based on ocaml 4.06.0+flambda, patched with https://github.com/ocaml/ocaml/pull/1340. The compiler is configured with `./configure -flambda -fPIC`
- **GNU C Compiler (GCC)** - gcc (Debian 6.3.0-18+deb9u1) 6.3.0 20170516
- **opam** - 1.2.2

Benchmarks are compiled on a separate machine and statically linked into a single standalone executable. The minimal amount of dynamically linked library referenced by the microbenchmarks are as follows:

```
# Obtained via [ldd benchmark.exe]
linux-vdso.so.1 (0x00007ffde6159000)
libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007f52be3f5000)
libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007f52be1f1000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f52bde52000)
/lib64/ld-linux-x86-64.so.2 (0x00007f52be997000)
```

Microbenchmarks that do not require is compiled directly using `ocamlopt.opt` in the modified compiler. Dependencies for other microbenchmarks (mostly in the evaluation section) are installed using `opam`, with `OCAMLPARAM="_,O3=1"`[1]. These evaluation microbenchmarks are compiled with `ocamlbuild` or `ocamlfind`. The installed dependency versions are as follows, obtained via `opam list` (The end of the list is pruned. The only real important thing here is the version and names of the package, not their descriptions):

```
# Installed packages for 4.06.0+fyp:
astring                       0.8.3  Alternative String module for OCaml
async                       v0.11.0  Monadic concurrency library
async_extra                 v0.11.0  Monadic concurrency library
async_kernel                v0.11.0  Monadic concurrency library
async_rpc_kernel            v0.11.0  Platform-independent core of Async RPC libr
async_unix                  v0.11.0  Monadic concurrency library
base                        v0.11.0  Full standard library replacement for OCaml
base-bigarray                  base  Bigarray library distributed with the OCaml
base-bytes                     base  Bytes library distributed with the OCaml co
base-raw_spacetime_lib         base  Raw_spacetime_lib library distributed with
base-threads                   base  Threads library distributed with the OCaml
base-unix                      base  Unix library distributed with the OCaml com
base64                       2.2.0  Base64 encoding for OCaml
bin_prot                    v0.11.0  A binary protocol generator
```

[1] OCaml allows the user to override compilation flags with environment variables

```
biniou                      1.2.0  Binary data format designed for speed, safe
camlimages                  5.0.1  Image processing library
camlzip                      1.07  Provides easy access to compressed files in
cmdliner                    1.0.2  Declarative definition of command line inte
cohttp                      1.0.0  An OCaml library for HTTP clients and serve
cohttp-lwt                  1.0.2  An OCaml library for HTTP clients and serve
cohttp-lwt-unix             1.0.2  An OCaml library for HTTP clients and serve
conduit                     1.1.0  An OCaml network connection establishment l
conduit-lwt                 1.1.0  An OCaml network connection establishment l
conduit-lwt-unix            1.1.0  An OCaml network connection establishment l
conf-autoconf                 0.1  Virtual package relying on autoconf install
conf-gmp                        1  Virtual package relying on a GMP lib system
conf-m4                         1  Virtual package relying on m4
conf-perl                       1  Virtual package relying on perl
conf-which                      1  Virtual package relying on which
configurator               v0.11.0  Helper library for gathering system configu
core                       v0.11.1  Industrial strength alternative to OCaml's
core_kernel                v0.11.0  Industrial strength alternative to OCaml's
cppo                         1.6.4  Equivalent of the C preprocessor for OCaml
depext                       1.0.5  Query and install external dependencies of
easy-format                  1.3.1  High-level and functional interface to the
fieldslib                  v0.11.0  Syntax extension to define first class valu
fmt                          0.8.5  OCaml Format pretty-printer combinators
ipaddr                       2.8.0  IP (and MAC) address manipulation
jane-street-headers        v0.11.0  Jane Street C header files
jbuilder                1.0+beta20  Fast, portable and opinionated build system
jsonm                        1.0.1  Non-blocking streaming JSON codec for OCaml
logs                         0.6.2  Logging infrastructure for OCaml
lwt                          3.3.0  Promises, concurrency, and parallelized I/O
magic-mime                   1.1.0  Map filenames to common MIME types
menhir                    20180530  LR(1) parser generator
num                             1.1  The legacy Num library for arbitrary-precis
ocaml-compiler-libs        v0.11.0  OCaml compiler libraries repackaged
ocaml-migrate-parsetree     1.0.11  Convert OCaml parsetrees between different
ocamlbuild                  0.12.0  OCamlbuild is a build system with builtin r
ocamlfind                    1.8.0  A library manager for OCaml
ocplib-endian                  1.0  Optimised functions to read and write int16
ocplib-simplex                 0.4  A library implementing a simplex algorithm,
octavius                     1.2.0  Ocamldoc comment syntax parser
parsexp                    v0.11.0  S-expression parsing library
ppx_assert                 v0.11.0  Assert-like extension nodes that raise usef
ppx_base                   v0.11.0  Base set of ppx rewriters
ppx_bench                  v0.11.0  Syntax extension for writing in-line benchm
ppx_bin_prot               v0.11.1  Generation of bin_prot readers and writers
ppx_compare                v0.11.1  Generation of comparison functions from typ
ppx_custom_printf          v0.11.0  Printf-style format-strings for user-define
ppx_derivers                    1.0  Shared [@@deriving] plugin registry
ppx_enumerate              v0.11.1  Generate a list containing all values of a
ppx_expect                 v0.11.0  Cram like framework for OCaml
ppx_fail                   v0.11.0  Add location to calls to failwiths
ppx_fields_conv            v0.11.0  Generation of accessor and iteration functi
ppx_hash                   v0.11.1  A ppx rewriter that generates hash function
ppx_here                   v0.11.0  Expands [%here] into its location
ppx_inline_test            v0.11.0  Syntax extension for writing in-line tests
ppx_jane                   v0.11.0  Standard Jane Street ppx rewriters
ppx_js_style               v0.11.0  Code style checker for Jane Street Packages
ppx_let                    v0.11.0  Monadic let-bindings
ppx_optcomp                v0.11.0  Optional compilation for OCaml
```

```
ppx_optional                v0.11.0  Pattern matching on flat options
ppx_pipebang                v0.11.0  A ppx rewriter that inlines reverse applica
ppx_sexp_conv               v0.11.2  Generation of S-expression conversion funct
ppx_sexp_message            v0.11.0  A ppx rewriter for easy construction of s-
e
ppx_sexp_value              v0.11.0  A ppx rewriter that simplifies building s-
e
ppx_tools_versioned             5.1  A variant of ppx_tools based on ocaml-
migra
ppx_typerep_conv            v0.11.1  Generation of runtime types from type decla
ppx_variants_conv           v0.11.1  Generation of accessor and iteration functi
ppxlib                        0.3.0  A comprehensive toolbox for ppx development
protocol_version_header     v0.11.0  Protocol versioning
psmt2-frontend                  0.1  A library to parse and type-check a conserv
re                            1.7.3  RE is a regular expression library for OCam
react                         1.2.1  Declarative events and signals for OCaml
result                          1.3  Compatibility Result module
sexplib                     v0.11.0  Library for serializing OCaml values to and
sexplib0                    v0.11.0  Library containing the definition of S-
expr
spawn                       v0.12.0  Spawning sub-processes
splittable_random           v0.11.0  PRNG that can be split into independent str
stdio                       v0.11.0  Standard IO library for OCaml
stringext                     1.5.0  Extra string functions for OCaml
topkg                         0.9.1  The transitory OCaml software packager
typerep                     v0.11.0  typerep is a library for runtime types.
uchar                         0.0.2  Compatibility library for OCaml's Uchar mod
uri                           1.9.6  An RFC3986 URI/URL parsing library
uuidm                         0.9.6  Universally unique identifiers (UUIDs) for
uutf                          1.0.1  Non-blocking streaming Unicode codec for OC
variantslib                 v0.11.0  Part of Jane Street's Core library
zarith                          1.7  Implements arithmetic and logical operation
```

D.2 Execution Environment

The 3 machines executing benchmarks have the following specifications:

Processor	intel i7 Haswell Quad-core processor
HyperThreading	Yes
Storage	500GB HDD
Memory	8 GB of DDR3 RAM
Operating System	Debian GNU/Linux 9 (stretch
Kernel Version	4.9.0-4-amd64
Boot Options	BOOT_IMAGE=/boot/vmlinuz-4.9.0-4-amd64 ro quiet isolcpus=0

Unless otherwise stated, benchmarks are executed using `taskset 0x1 ./binary.exe arg1 arg2` without the interference of other programs. Besides the benchmark program, the only other user-space programs running are:

```
/lib/systemd/systemd
/sbin/agentty
/usr/sbin/sshd
/usr/bin/dbus-daemon
/usr/sbin/cron
/usr/sbin/rsyslog
/lib/systemd/systemd-logind
/lib/systemd/systemd-timesyncd
/lib/systemd/systemd-udevd
```

```
/lib/systemd/systemd-journald
```

Appendix E

Detailed Results

E.1 Predictive Modelling

Benchmark	*Best Seen*	$h*$		$h^{(0)}_{general}$		$h^{(1)}_{general}$	
		Speedup	Q	Speedup	Q	Speedup	Q
almabench	*3.879%*	*1.762%*	*(0.546)*	*1.762%*	*(0.546)*	*1.443%*	*(0.628)*
		3.024%	*(0.220)*	*0.771%*	*(0.801)*	*1.857%*	*(0.521)*
bdd	*3.701%*	*3.956%*	*(-0.069)*	*3.327%*	*(0.101)*	*3.721%*	*(-0.005)*
		4.380%	*(-0.183)*	*3.731%*	*(-0.008)*	*2.214%*	*(0.402)*
fft	*3.170%*	*2.705%*	*(0.147)*	*1.700%*	*(0.464)*	*2.319%*	*(0.268)*
		2.920%	*(0.079)*	*2.640%*	*(0.167)*	*2.176%*	*(0.313)*
floats-in-functor	*54.954%*	*55.328%*	*(-0.007)*	*53.342%*	*(0.029)*	*55.208%*	*(-0.005)*
		55.346%	*(-0.007)*	*55.236%*	*(-0.005)*	*55.229%*	*(-0.005)*
hamming	*4.523%*	*4.713%*	*(-0.042)*	*3.460%*	*(0.235)*	*4.188%*	*(0.074)*
		3.534%	*(0.219)*	*3.526%*	*(0.220)*	*2.606%*	*(0.424)*
kahan-sum	*1.430%*	*1.472%*	*(-0.029)*	*1.415%*	*(0.011)*	*1.376%*	*(0.037)*
		1.514%	*(-0.059)*	*1.418%*	*(0.008)*	*1.365%*	*(0.045)*
kb	*9.147%*	*8.978%*	*(0.018)*	*8.651%*	*(0.054)*	*4.059%*	*(0.556)*
		10.093%	*(-0.103)*	*10.061%*	*(-0.100)*	*6.158%*	*(0.327)*
lens	*54.429%*	*54.645%*	*(-0.004)*	*54.539%*	*(-0.002)*	*54.210%*	*(0.004)*
		54.648%	*(-0.004)*	*54.203%*	*(0.004)*	*54.210%*	*(0.004)*
lexifi	*4.774%*	*5.027%*	*(-0.053)*	*4.411%*	*(0.076)*	*4.736%*	*(0.008)*
		5.307%	*(-0.112)*	*5.201%*	*(-0.089)*	*5.138%*	*(-0.076)*
quicksort	*1.532%*	*1.900%*	*(-0.240)*	*1.572%*	*(-0.027)*	*1.164%*	*(0.240)*
		1.627%	*(-0.062)*	*1.245%*	*(0.187)*	*0.977%*	*(0.362)*
sequence	*5.372%*	*5.227%*	*(0.027)*	*3.550%*	*(0.339)*	*3.809%*	*(0.291)*
		5.693%	*(-0.060)*	*0.170%*	*(0.968)*	*5.693%*	*(-0.060)*
sequence-cps	*0.144%*	*0.295%*	*(-1.044)*	*0.264%*	*(-0.834)*	*0.290%*	*(-1.012)*
		0.301%	*(-1.093)*	*0.290%*	*(-1.012)*	*0.292%*	*(-1.028)*

Table E.1: Results of "optimal" decisions generated from predictive modelling. The first row in every benchmark denotes those obtained with L1 regularisation, and the second row denotes those obtained from L2 regularisation. Q is the reconstruction quality (lower better), and speedup is the speed up over baseline (higher better). h^* denotes cases where hyperparameters are chosen individually for experiments, whereas $h^{(0)}_{general}$ and $h^{(1)}_{general}$ are general-purpose hyperparamters chosen over by minimising the total reconstruction quality (defined in 6.9)

Benchmark	γ	λ	$f_{preprocess}$
almabench	1.000000	0.050000	linear_speedup_over_mean
bdd	0.400000	0.500000	tanh_speedup_over_baseline
fft	1.000000	0.050000	log_speedup_over_mean
floats-in-functor	0.900000	0.500000	log_speedup_over_baseline
hamming	0.400000	0.500000	linear_speedup_over_baseline
kahan-sum	1.000000	2.000000	log_speedup_over_mean
kb	0.400000	0.500000	log_speedup_over_baseline
lens	1.000000	0.050000	tanh_speedup_over_baseline
lexifi	1.000000	0.500000	linear_speedup_over_baseline
quicksort	0.900000	0.050000	log_speedup_over_mean
sequence	0.400000	0.050000	log_speedup_over_mean
sequence-cps	0.400000	0.500000	linear_speedup_over_mean

Table E.2: Individually tuned hyperparameter values when using predicting modelling with L2 regularisation

Benchmark	γ	$f_{preprocess}$
almabench	0.400000	linear_speedup_over_baseline
bdd	1.000000	tanh_speedup_over_mean
fft	0.990000	log_speedup_over_mean
floats-in-functor	1.000000	linear_speedup_over_baseline
hamming	0.990000	log_speedup_over_mean
kahan-sum	1.000000	linear_speedup_over_baseline
kb	0.990000	log_speedup_over_mean
lens	0.400000	tanh_speedup_over_mean
lexifi	0.900000	tanh_speedup_over_mean
quicksort	0.400000	tanh_speedup_over_baseline
sequence	1.000000	tanh_speedup_over_mean
sequence-cps	0.990000	tanh_speedup_over_baseline

Table E.3: Individually tuned hyperparameter values when using predicting modelling with L1 regularisation (Note that λ is automatically tuned using cross validation)

E.2 Inlining Policies

A statistical summary of the speedup obtained by different inlining policies throughout the training and validation benchmark. The headers labelled as R, N and I refers to the number of regressions, "No change" and improvements due the the model.

Reg	Model	τ	25-th	median	75-th	min	max	R	N	I
L1	$h_{general}$	-	0.464%	1.770%	3.3538%	-4.308%	16.676%	1	7	10
L1	h_{hand}	-	-0.853%	0.417%	1.4993%	-8.118%	31.449%	4	8	6
L1	H_*	-	-0.995%	0.959%	2.1857%	-4.387%	8.812%	4	6	8
L2	$h_{general}$	0.000050	-0.197%	4.766%	8.2498%	-6.548%	28.865%	3	3	12
L2	$h_{general}$	0.000100	-0.530%	1.384%	5.2366%	-2.939%	32.756%	2	7	9
L2	$h_{general}$	0.000500	0.026%	1.365%	4.6891%	-8.312%	34.630%	1	6	11
L2	$h_{general}$	0.001000	-0.690%	0.515%	4.6346%	-10.928%	19.347%	4	8	6
L2	$h_{general}$	0.005000	-0.964%	0.903%	2.8015%	-5.935%	13.879%	3	7	8
L2	$h_{general}$	0.010000	-0.106%	0.490%	1.5678%	-7.725%	31.740%	2	10	6
L2	$h_{general}$	0.050000	-0.018%	0.340%	0.7940%	-3.568%	8.568%	2	12	4
L2	h_{hand}	0.000050	-0.330%	1.730%	2.8326%	-3.024%	19.102%	3	4	11
L2	h_{hand}	0.000100	-0.547%	1.431%	4.3959%	-23.517%	18.450%	4	4	10
L2	h_{hand}	0.000500	0.365%	1.878%	5.1664%	-18.490%	17.397%	3	3	12
L2	h_{hand}	0.001000	-0.050%	0.910%	3.0457%	-5.786%	34.960%	2	8	8
L2	h_{hand}	0.005000	-0.220%	1.010%	1.9816%	-5.418%	14.152%	2	7	9
L2	h_{hand}	0.010000	-0.040%	0.399%	2.2163%	-5.016%	16.621%	3	8	7
L2	h_{hand}	0.050000	-0.962%	-0.087%	0.8124%	-9.418%	11.258%	4	11	3
L2	H_*	0.000050	0.364%	1.875%	6.9024%	-7.388%	9.202%	1	6	11
L2	H_*	0.000100	-0.133%	1.652%	6.8754%	-7.465%	15.050%	3	5	10
L2	H_*	0.000500	-0.523%	2.033%	5.1652%	-8.174%	7.918%	4	3	11
L2	H_*	0.001000	0.067%	1.501%	2.7280%	-3.122%	7.053%	2	7	9
L2	H_*	0.005000	0.035%	1.407%	7.0739%	-8.288%	16.875%	1	8	9
L2	H_*	0.010000	-0.618%	0.696%	1.3152%	-5.311%	17.869%	3	8	7
L2	H_*	0.050000	-0.506%	0.023%	0.3983%	-8.009%	2.598%	3	13	2

The table below shows the exhaustive list of results from running benchmarks:

Benchmark	**Reward Model**		τ	**Time(s)**	**Speedup**
almabench	Baseline			10.730	0.000%
	L2	h_{hand}	0.001000	10.449	2.620%
	L2	H_*	0.000500	10.495	2.191%
	L2	H_*	0.000050	10.544	1.733%
	L2	$h_{general}$	0.000500	10.637	0.873%
	L2	h_{hand}	0.005000	10.648	0.762%
	L2	CMoE	-	10.656	0.692%
	L2	H_*	0.000100	10.660	0.655%
	L2	h_{hand}	0.000100	10.673	0.534%
	L1	H_*	-	10.690	0.374%
	L1	h_{hand}	-	10.695	0.325%
	L2	H_*	0.050000	10.705	0.237%
	L2	$h_{general}$	0.010000	10.708	0.205%
	L2	$h_{general}$	0.000100	10.709	0.196%
	L2	h_{hand}	0.000050	10.733	-0.022%
	L2	$h_{general}$	0.050000	10.738	-0.071%
	L2	h_{hand}	0.010000	10.754	-0.224%
	L2	$h_{general}$	0.000050	10.772	-0.386%
	L2	H_*	0.001000	10.797	-0.626%
	L1	CMoE	-	10.803	-0.673%
	L2	h_{hand}	0.000500	10.805	-0.695%
	L1	$h_{general}$	-	10.825	-0.879%
	L2	H_*	0.005000	10.831	-0.935%

	L2	$h_{general}$	0.005000	10.849	-1.107%
	L2	H_*	0.010000	10.849	-1.111%
	L2	h_{hand}	0.050000	10.856	-1.168%
	L2	$h_{general}$	0.001000	10.893	-1.516%
lexifi		Baseline		15.237	0.000%
	L2	h_{hand}	0.000500	14.547	4.529%
	L2	$h_{general}$	0.000100	14.551	4.501%
	L2	$h_{general}$	0.000050	14.559	4.452%
	L2	h_{hand}	0.000100	14.567	4.398%
	L2	H_*	0.000050	14.570	4.381%
	L2	h_{hand}	0.001000	14.584	4.290%
	L2	H_*	0.001000	14.589	4.254%
	L2	h_{hand}	0.000050	14.604	4.153%
	L2	h_{hand}	0.010000	14.638	3.932%
	L2	H_*	0.000500	14.660	3.789%
	L2	$h_{general}$	0.010000	14.687	3.611%
	L2	H_*	0.000100	14.714	3.431%
	L2	$h_{general}$	0.000500	14.814	2.776%
	L2	CMoE	-	14.823	2.715%
	L2	$h_{general}$	0.050000	14.994	1.597%
	L1	h_{hand}	-	15.002	1.541%
	L1	$h_{general}$	-	15.019	1.431%
	L2	$h_{general}$	0.005000	15.038	1.307%
	L1	H_*	-	15.039	1.302%
	L2	$h_{general}$	0.001000	15.085	1.000%
	L1	CMoE	-	15.087	0.989%
	L2	H_*	0.010000	15.263	-0.167%
	L2	H_*	0.005000	15.273	-0.233%
	L2	h_{hand}	0.050000	15.295	-0.380%
	L2	H_*	0.050000	15.325	-0.579%
	L2	h_{hand}	0.005000	15.533	-1.941%
sequence		Baseline		12.251	0.000%
	L2	H_*	0.005000	11.405	6.905%
	L2	h_{hand}	0.000100	11.481	6.286%
	L2	H_*	0.000500	11.664	4.792%
	L2	$h_{general}$	0.001000	11.812	3.584%
	L2	h_{hand}	0.010000	11.812	3.581%
	L2	H_*	0.000050	11.897	2.887%
	L2	h_{hand}	0.001000	11.905	2.823%
	L2	H_*	0.001000	11.920	2.702%
	L2	$h_{general}$	0.000500	11.931	2.611%
	L1	CMoE	-	11.962	2.363%
	L2	h_{hand}	0.000050	11.977	2.238%
	L2	$h_{general}$	0.005000	12.154	0.793%
	L2	$h_{general}$	0.000050	12.175	0.624%
	L1	$h_{general}$	-	12.176	0.612%
	L2	h_{hand}	0.050000	12.239	0.096%
	L1	H_*	-	12.250	0.014%
	L2	H_*	0.000100	12.251	0.004%
	L2	H_*	0.010000	12.252	-0.006%
	L2	$h_{general}$	0.010000	12.253	-0.016%
	L2	H_*	0.050000	12.262	-0.089%
	L2	CMoE	-	12.317	-0.541%
	L1	h_{hand}	-	12.400	-1.218%
	L2	h_{hand}	0.005000	12.418	-1.362%
	L2	$h_{general}$	0.050000	12.691	-3.588%
	L2	h_{hand}	0.000500	12.721	-3.833%
	L2	$h_{general}$	0.000100	12.770	-4.231%

floats-in-functor		Baseline		9.320	0.000%
	L2	H_*	0.000500	4.254	54.359%
	L2	H_*	0.000100	4.258	54.316%
	L2	h_{hand}	0.000100	4.260	54.294%
	L2	h_{hand}	0.000050	4.265	54.235%
	L2	H_*	0.000050	4.305	53.812%
	L2	h_{hand}	0.001000	4.305	53.810%
	L2	h_{hand}	0.010000	4.306	53.795%
	L2	$h_{general}$	0.000050	4.307	53.792%
	L2	H_*	0.001000	4.310	53.751%
	L2	$h_{general}$	0.001000	4.314	53.715%
	L2	$h_{general}$	0.000100	4.322	53.626%
	L2	h_{hand}	0.005000	4.390	52.897%
	L2	h_{hand}	0.000500	4.394	52.858%
	L2	$h_{general}$	0.000500	4.394	52.853%
	L2	$h_{general}$	0.005000	4.397	52.818%
	L2	h_{hand}	0.050000	4.398	52.815%
	L1	H_*	-	4.569	50.974%
	L2	$h_{general}$	0.010000	4.572	50.949%
	L2	$h_{general}$	0.050000	4.578	50.880%
	L2	CMoE	-	4.580	50.864%
	L1	h_{hand}	-	4.752	49.016%
	L1	CMoE	-	4.755	48.979%
	L2	H_*	0.050000	6.310	32.296%
	L1	$h_{general}$	-	6.310	32.296%
	L2	H_*	0.005000	6.317	32.226%
	L2	H_*	0.010000	9.323	-0.034%
fft		Baseline		8.151	0.000%
	L2	h_{hand}	0.000050	7.961	2.331%
	L2	$h_{general}$	0.010000	7.971	2.213%
	L2	h_{hand}	0.001000	7.976	2.158%
	L2	$h_{general}$	0.000100	7.978	2.127%
	L2	h_{hand}	0.010000	7.981	2.093%
	L2	$h_{general}$	0.005000	7.982	2.080%
	L2	h_{hand}	0.005000	7.983	2.067%
	L2	$h_{general}$	0.000500	7.984	2.049%
	L2	CMoE	-	7.989	1.992%
	L2	h_{hand}	0.000500	7.993	1.949%
	L1	H_*	-	7.993	1.939%
	L2	$h_{general}$	0.000050	8.001	1.845%
	L2	h_{hand}	0.000100	8.003	1.824%
	L2	H_*	0.000500	8.012	1.709%
	L2	h_{hand}	0.050000	8.012	1.709%
	L2	$h_{general}$	0.050000	8.019	1.630%
	L2	$h_{general}$	0.001000	8.028	1.518%
	L2	H_*	0.001000	8.035	1.427%
	L2	H_*	0.000100	8.037	1.404%
	L2	H_*	0.050000	8.040	1.369%
	L2	H_*	0.005000	8.048	1.272%
	L2	H_*	0.000050	8.054	1.193%
	L1	CMoE	-	8.056	1.167%
	L2	H_*	0.010000	8.058	1.148%
	L1	h_{hand}	-	8.061	1.114%
	L1	$h_{general}$	-	8.072	0.977%
nucleic		Baseline		6.449	0.000%
	L2	$h_{general}$	0.001000	5.794	10.167%
	L2	H_*	0.000050	5.856	9.202%

	L2	H_*	0.000500	5.974	7.375%
	L2	$h_{general}$	0.000050	6.032	6.473%
	L2	CMoE	-	6.034	6.444%
	L2	$h_{general}$	0.000100	6.112	5.237%
	L2	H_*	0.000100	6.116	5.172%
	L2	$h_{general}$	0.050000	6.254	3.028%
	L2	H_*	0.001000	6.274	2.728%
	L2	h_{hand}	0.000500	6.295	2.401%
	L2	H_*	0.005000	6.332	1.817%
	L2	$h_{general}$	0.010000	6.388	0.947%
	L2	h_{hand}	0.005000	6.391	0.901%
	L2	H_*	0.010000	6.406	0.678%
	L1	$h_{general}$	-	6.415	0.538%
	L2	$h_{general}$	0.000500	6.432	0.266%
	L2	h_{hand}	0.001000	6.434	0.246%
	L2	$h_{general}$	0.005000	6.437	0.200%
	L2	H_*	0.050000	6.466	-0.260%
	L2	h_{hand}	0.050000	6.479	-0.452%
	L2	h_{hand}	0.000050	6.492	-0.664%
	L2	h_{hand}	0.010000	6.515	-1.013%
	L1	H_*	-	6.519	-1.071%
	L1	CMoE	-	6.541	-1.414%
	L2	h_{hand}	0.000100	6.550	-1.562%
	L1	h_{hand}	-	6.567	-1.822%
chameneos-redux-lwt		Baseline		6.428	0.000%
	L2	H_*	0.000050	6.023	6.303%
	L2	h_{hand}	0.000100	6.160	4.172%
	L1	h_{hand}	-	6.162	4.145%
	L2	$h_{general}$	0.005000	6.201	3.534%
	L2	h_{hand}	0.000050	6.246	2.833%
	L2	H_*	0.050000	6.261	2.598%
	L2	H_*	0.000100	6.268	2.485%
	L2	h_{hand}	0.001000	6.296	2.061%
	L2	$h_{general}$	0.000100	6.307	1.880%
	L2	h_{hand}	0.000500	6.307	1.878%
	L2	H_*	0.001000	6.312	1.809%
	L1	$h_{general}$	-	6.316	1.742%
	L2	H_*	0.000500	6.319	1.691%
	L2	CMoE	-	6.363	1.011%
	L2	h_{hand}	0.010000	6.398	0.476%
	L1	H_*	-	6.406	0.351%
	L2	$h_{general}$	0.050000	6.407	0.334%
	L2	H_*	0.005000	6.426	0.035%
	L2	h_{hand}	0.005000	6.442	-0.220%
	L1	CMoE	-	6.458	-0.462%
	L2	$h_{general}$	0.000050	6.512	-1.310%
	L2	H_*	0.010000	6.554	-1.965%
	L2	$h_{general}$	0.010000	6.720	-4.545%
	L2	$h_{general}$	0.001000	6.747	-4.966%
	L2	$h_{general}$	0.000500	6.962	-8.312%
	L2	h_{hand}	0.050000	7.034	-9.418%
list-benchmark		Baseline		5.176	0.000%
	L2	$h_{general}$	0.000050	4.862	6.069%
	L2	h_{hand}	0.000100	4.877	5.776%
	L2	h_{hand}	0.000050	4.894	5.459%
	L2	H_*	0.000500	4.909	5.165%
	L2	$h_{general}$	0.000500	4.934	4.689%
	L2	$h_{general}$	0.001000	4.937	4.635%

	L1	$h_{general}$	-	5.003	3.354%
	L2	CMoE	-	5.008	3.248%
	L2	H_*	0.001000	5.018	3.061%
	L2	h_{hand}	0.001000	5.019	3.046%
	L2	$h_{general}$	0.000100	5.022	2.984%
	L2	H_*	0.005000	5.029	2.854%
	L2	H_*	0.000100	5.056	2.318%
	L2	h_{hand}	0.010000	5.062	2.216%
	L1	H_*	-	5.063	2.186%
	L2	$h_{general}$	0.005000	5.065	2.146%
	L2	h_{hand}	0.005000	5.074	1.982%
	L2	H_*	0.000050	5.094	1.594%
	L2	$h_{general}$	0.010000	5.095	1.568%
	L2	H_*	0.010000	5.106	1.357%
	L2	h_{hand}	0.000500	5.113	1.230%
	L2	$h_{general}$	0.050000	5.113	1.227%
	L2	h_{hand}	0.050000	5.128	0.927%
	L1	h_{hand}	-	5.137	0.760%
	L1	CMoE	-	5.159	0.339%
	L2	H_*	0.050000	5.206	-0.563%
chameneos-redux-async		Baseline		5.328	0.000%
	L2	H_*	0.000050	5.265	1.191%
	L2	$h_{general}$	0.010000	5.317	0.219%
	L2	H_*	0.005000	5.319	0.168%
	L2	$h_{general}$	0.000500	5.340	-0.227%
	L2	h_{hand}	0.000050	5.383	-1.028%
	L1	H_*	-	5.387	-1.105%
	L2	h_{hand}	0.005000	5.403	-1.395%
	L2	H_*	0.010000	5.420	-1.713%
	L2	h_{hand}	0.010000	5.443	-2.156%
	L2	H_*	0.001000	5.484	-2.920%
	L2	$h_{general}$	0.000100	5.485	-2.939%
	L2	H_*	0.050000	5.503	-3.268%
	L2	$h_{general}$	0.050000	5.518	-3.568%
	L1	h_{hand}	-	5.521	-3.617%
	L2	h_{hand}	0.000100	5.524	-3.663%
	L2	H_*	0.000500	5.555	-4.255%
	L1	$h_{general}$	-	5.558	-4.308%
	L2	h_{hand}	0.050000	5.569	-4.509%
	L2	$h_{general}$	0.001000	5.585	-4.813%
	L1	CMoE	-	5.586	-4.833%
	L2	h_{hand}	0.001000	5.623	-5.536%
	L2	CMoE	-	5.639	-5.826%
	L2	$h_{general}$	0.005000	5.645	-5.935%
	L2	$h_{general}$	0.000050	5.677	-6.548%
	L2	H_*	0.000100	5.682	-6.631%
	L2	h_{hand}	0.000500	6.314	-18.490%
kahan-sum		Baseline		8.774	-0.381%
	L2	h_{hand}	0.000050	6.651	23.908%
	L2	H_*	0.000100	7.245	17.109%
	L2	$h_{general}$	0.000050	7.250	17.059%
	L2	H_*	0.000050	7.262	16.921%
	L2	CMoE	-	8.048	7.925%
	L2	$h_{general}$	0.000100	8.072	7.655%
	L2	h_{hand}	0.001000	8.099	7.345%
	L2	H_*	0.000500	8.130	6.985%
	L2	h_{hand}	0.010000	8.197	6.220%
	L2	$h_{general}$	0.000500	8.224	5.911%

	L2	h_{hand}	0.000500	8.255	5.556%
	L2	h_{hand}	0.005000	8.315	4.875%
	L1	H_*	-	8.541	2.286%
	L2	H_*	0.001000	8.567	1.993%
	L1	CMoE	-	8.585	1.790%
	L1	h_{hand}	-	8.610	1.503%
	L2	h_{hand}	0.000100	8.669	0.827%
	L1	$h_{general}$	-	8.681	0.690%
	L2	H_*	0.010000	8.707	0.387%
	L2	$h_{general}$	0.001000	8.754	-0.148%
	L2	$h_{general}$	0.050000	8.759	-0.212%
	L2	H_*	0.050000	8.793	-0.593%
	L2	h_{hand}	0.050000	8.807	-0.753%
	L2	$h_{general}$	0.005000	8.813	-0.821%
	L2	H_*	0.005000	8.816	-0.862%
	L2	$h_{general}$	0.010000	8.892	-1.728%
vector-functor		Baseline		8.854	0.000%
	L2	H_*	0.000100	8.811	0.487%
	L2	h_{hand}	0.000050	8.818	0.405%
	L2	h_{hand}	0.000500	8.822	0.365%
	L2	$h_{general}$	0.050000	8.851	0.038%
	L2	$h_{general}$	0.000500	8.852	0.026%
	L2	H_*	0.050000	8.853	0.011%
	L1	$h_{general}$	-	8.854	0.004%
	L2	CMoE	-	8.855	-0.006%
	L1	CMoE	-	8.857	-0.036%
	L2	h_{hand}	0.010000	8.858	-0.040%
	L2	H_*	0.001000	8.858	-0.042%
	L2	H_*	0.005000	8.860	-0.065%
	L1	h_{hand}	-	8.864	-0.107%
	L2	$h_{general}$	0.000050	8.896	-0.474%
	L2	h_{hand}	0.005000	8.900	-0.518%
	L2	H_*	0.000500	8.901	-0.523%
	L2	$h_{general}$	0.000100	8.901	-0.530%
	L2	h_{hand}	0.000100	8.903	-0.547%
	L2	h_{hand}	0.001000	8.910	-0.633%
	L2	H_*	0.000050	8.933	-0.890%
	L1	H_*	-	8.938	-0.940%
	L2	$h_{general}$	0.010000	8.938	-0.947%
	L2	h_{hand}	0.050000	8.939	-0.962%
	L2	H_*	0.010000	8.941	-0.984%
	L2	$h_{general}$	0.005000	8.942	-0.988%
	L2	$h_{general}$	0.001000	8.947	-1.048%
bdd		Baseline		13.117	0.000%
	L2	H_*	0.001000	12.519	4.557%
	L2	H_*	0.000050	12.538	4.413%
	L2	H_*	0.050000	12.564	4.218%
	L2	CMoE	-	12.569	4.175%
	L2	$h_{general}$	0.000100	12.585	4.055%
	L2	$h_{general}$	0.001000	12.627	3.735%
	L1	h_{hand}	-	12.665	3.447%
	L2	h_{hand}	0.005000	12.726	2.977%
	L2	H_*	0.000500	12.775	2.607%
	L2	h_{hand}	0.010000	12.927	1.444%
	L2	h_{hand}	0.000050	12.960	1.199%
	L2	H_*	0.010000	12.982	1.031%
	L2	$h_{general}$	0.000050	12.987	0.991%
	L2	$h_{general}$	0.010000	13.013	0.789%

	L2	h_{hand}	0.000100	13.023	0.714%
	L2	h_{hand}	0.001000	13.036	0.613%
	L2	H_*	0.005000	13.049	0.519%
	L2	$h_{general}$	0.005000	13.053	0.486%
	L2	$h_{general}$	0.000500	13.085	0.239%
	L2	H_*	0.000100	13.091	0.195%
	L1	H_*	-	13.142	-0.196%
	L2	$h_{general}$	0.050000	13.159	-0.320%
	L2	h_{hand}	0.050000	13.211	-0.720%
	L2	h_{hand}	0.000500	13.710	-4.525%
	L1	$h_{general}$	-	13.760	-4.907%
	L1	CMoE	-	13.801	-5.214%
valet-lwt		Baseline		5.650	0.000%
	L2	$h_{general}$	0.000100	5.151	8.827%
	L2	h_{hand}	0.000100	5.159	8.702%
	L2	H_*	0.005000	5.163	8.630%
	L2	$h_{general}$	0.050000	5.166	8.568%
	L2	H_*	0.000050	5.174	8.433%
	L1	CMoE	-	5.174	8.419%
	L2	H_*	0.000100	5.179	8.338%
	L2	$h_{general}$	0.000050	5.180	8.314%
	L1	$h_{general}$	-	5.187	8.199%
	L2	h_{hand}	0.000500	5.201	7.956%
	L2	H_*	0.000500	5.203	7.918%
	L2	CMoE	-	5.224	7.545%
	L2	h_{hand}	0.000050	5.581	1.224%
	L2	$h_{general}$	0.001000	5.597	0.934%
	L2	h_{hand}	0.001000	5.599	0.910%
	L1	H_*	-	5.601	0.863%
	L2	H_*	0.010000	5.611	0.696%
	L2	h_{hand}	0.005000	5.621	0.515%
	L2	$h_{general}$	0.010000	5.622	0.490%
	L2	h_{hand}	0.010000	5.628	0.399%
	L2	$h_{general}$	0.000500	5.636	0.259%
	L2	H_*	0.001000	5.636	0.253%
	L1	h_{hand}	-	5.658	-0.141%
	L2	H_*	0.050000	5.679	-0.506%
	L2	h_{hand}	0.050000	5.700	-0.877%
	L2	$h_{general}$	0.005000	5.705	-0.964%
nullable-array		Baseline		6.258	0.000%
	L2	$h_{general}$	0.005000	5.389	13.879%
	L2	h_{hand}	0.001000	5.437	13.120%
	L2	H_*	0.000100	5.710	8.751%
	L2	$h_{general}$	0.000050	5.742	8.250%
	L2	h_{hand}	0.000050	5.744	8.208%
	L2	h_{hand}	0.000100	5.757	8.006%
	L2	h_{hand}	0.010000	5.759	7.977%
	L2	$h_{general}$	0.001000	5.765	7.874%
	L2	H_*	0.010000	5.765	7.870%
	L2	h_{hand}	0.050000	5.775	7.719%
	L2	$h_{general}$	0.000500	5.780	7.634%
	L1	H_*	-	5.782	7.606%
	L2	h_{hand}	0.005000	5.797	7.357%
	L2	H_*	0.005000	5.800	7.310%
	L2	CMoE	-	5.805	7.230%
	L2	H_*	0.001000	5.816	7.053%
	L2	H_*	0.000500	5.829	6.853%
	L2	h_{hand}	0.000500	5.935	5.166%

	L2	$h_{general}$	0.010000	5.990	4.287%
	L2	H_*	0.000050	6.115	2.281%
	L1	$h_{general}$	-	6.147	1.770%
	L1	CMoE	-	6.185	1.156%
	L2	$h_{general}$	0.050000	6.248	0.155%
	L2	H_*	0.050000	6.251	0.101%
	L2	$h_{general}$	0.000100	6.259	-0.016%
	L1	h_{hand}	-	6.282	-0.380%
quicksort		Baseline		13.889	0.000%
	L1	$h_{general}$	-	13.694	1.410%
	L2	H_*	0.000500	13.694	1.405%
	L2	H_*	0.000050	13.700	1.362%
	L2	$h_{general}$	0.001000	13.703	1.345%
	L2	h_{hand}	0.000050	13.728	1.164%
	L2	h_{hand}	0.000500	13.744	1.045%
	L1	CMoE	-	13.744	1.044%
	L2	h_{hand}	0.001000	13.754	0.975%
	L1	h_{hand}	-	13.757	0.957%
	L1	H_*	-	13.761	0.928%
	L2	$h_{general}$	0.000050	13.762	0.920%
	L2	$h_{general}$	0.010000	13.767	0.883%
	L2	H_*	0.001000	13.767	0.883%
	L2	h_{hand}	0.010000	13.770	0.862%
	L2	h_{hand}	0.000100	13.770	0.862%
	L2	CMoE	-	13.771	0.853%
	L2	H_*	0.005000	13.779	0.793%
	L2	$h_{general}$	0.005000	13.780	0.791%
	L2	h_{hand}	0.005000	13.797	0.665%
	L2	H_*	0.000100	13.880	0.066%
	L2	H_*	0.050000	13.891	-0.013%
	L2	h_{hand}	0.050000	13.895	-0.040%
	L2	H_*	0.010000	13.895	-0.043%
	L2	$h_{general}$	0.050000	13.896	-0.046%
	L2	$h_{general}$	0.000100	13.899	-0.070%
	L2	$h_{general}$	0.000500	13.901	-0.083%
fyq-rev-list		Baseline		1.626	0.000%
	L2	$h_{general}$	0.000050	1.621	0.254%
	L2	h_{hand}	0.000050	1.623	0.148%
	L2	H_*	0.000100	1.624	0.103%
	L2	h_{hand}	0.050000	1.626	-0.045%
	L2	$h_{general}$	0.005000	1.626	-0.045%
	L2	H_*	0.001000	1.627	-0.066%
	L2	H_*	0.005000	1.627	-0.066%
	L1	h_{hand}	-	1.627	-0.090%
	L2	$h_{general}$	0.010000	1.629	-0.240%
	L2	H_*	0.050000	1.632	-0.428%
	L2	$h_{general}$	0.050000	1.634	-0.491%
	L2	H_*	0.010000	2.537	-56.092%
	L2	H_*	0.000500	2.540	-56.229%
	L2	h_{hand}	0.005000	2.542	-56.408%
	L1	H_*	-	2.545	-56.549%
	L2	$h_{general}$	0.000500	2.554	-57.093%
	L2	h_{hand}	0.000100	2.554	-57.122%
	L1	$h_{general}$	-	2.555	-57.171%
	L1	CMoE	-	2.564	-57.728%
	L2	CMoE	-	13.622	-737.989%
	L2	h_{hand}	0.010000	13.636	-738.861%
	L2	H_*	0.000050	13.714	-743.688%

	L2	h_{hand}	0.001000	13.740	-745.269%
	L2	$h_{general}$	0.001000	13.761	-746.526%
	L2	h_{hand}	0.000500	14.166	-771.482%
	L2	$h_{general}$	0.000100	14.206	-773.932%
naive-multilayer		Baseline		2.627	0.000%
	L2	$h_{general}$	0.000050	2.097	20.191%
	L2	$h_{general}$	0.001000	2.119	19.347%
	L2	h_{hand}	0.000050	2.125	19.102%
	L2	h_{hand}	0.000100	2.143	18.450%
	L2	CMoE	-	2.162	17.693%
	L2	h_{hand}	0.001000	2.198	16.327%
	L2	H_*	0.005000	2.293	12.723%
	L2	H_*	0.010000	2.313	11.949%
	L2	H_*	0.000100	2.330	11.330%
	L2	h_{hand}	0.050000	2.331	11.258%
	L2	h_{hand}	0.000500	2.359	10.201%
	L2	$h_{general}$	0.000100	2.365	9.972%
	L2	$h_{general}$	0.010000	2.372	9.718%
	L1	$h_{general}$	-	2.372	9.703%
	L1	CMoE	-	2.379	9.456%
	L2	$h_{general}$	0.000500	2.387	9.129%
	L1	H_*	-	2.396	8.812%
	L2	h_{hand}	0.010000	2.397	8.781%
	L1	h_{hand}	-	2.399	8.701%
	L2	$h_{general}$	0.005000	2.412	8.178%
	L2	H_*	0.000050	2.417	7.998%
	L2	H_*	0.000500	2.437	7.234%
	L2	h_{hand}	0.005000	2.535	3.496%
	L2	$h_{general}$	0.050000	2.663	-1.368%
	L2	H_*	0.001000	2.709	-3.122%
	L2	H_*	0.050000	2.722	-3.597%
sequence-cps		Baseline		14.251	0.000%
	L1	h_{hand}	-	14.251	0.006%
	L1	H_*	-	14.251	0.004%
	L2	$h_{general}$	0.010000	14.251	0.003%
	L2	h_{hand}	0.005000	14.251	0.003%
	L2	H_*	0.001000	14.251	0.003%
	L2	h_{hand}	0.050000	14.251	0.003%
	L2	H_*	0.050000	14.251	0.003%
	L2	$h_{general}$	0.005000	14.251	0.003%
	L2	H_*	0.010000	14.251	0.002%
	L2	h_{hand}	0.010000	14.251	0.002%
	L2	h_{hand}	0.000500	14.251	0.002%
	L2	H_*	0.005000	14.251	0.001%
	L2	$h_{general}$	0.050000	14.251	0.001%
	L2	CMoE	-	14.252	-0.001%
	L2	H_*	0.000050	14.252	-0.002%
	L2	$h_{general}$	0.000050	14.252	-0.003%
	L2	$h_{general}$	0.000100	14.252	-0.004%
	L2	$h_{general}$	0.001000	14.252	-0.007%
	L2	h_{hand}	0.000100	14.254	-0.016%
	L2	$h_{general}$	0.000500	14.255	-0.022%
	L1	CMoE	-	14.257	-0.038%
	L1	$h_{general}$	-	14.258	-0.044%
	L2	h_{hand}	0.000050	14.259	-0.051%
	L2	H_*	0.000500	14.260	-0.062%
	L2	H_*	0.000100	14.261	-0.069%
	L2	h_{hand}	0.001000	14.263	-0.078%

levinson-durbin		Baseline		4.873	0.000%
	L1	$h_{general}$	-	4.733	2.871%
	L2	h_{hand}	0.001000	4.746	2.606%
	L2	$h_{general}$	0.005000	4.787	1.766%
	L2	CMoE	-	4.788	1.751%
	L2	h_{hand}	0.010000	4.790	1.700%
	L2	H_*	0.000100	4.801	1.467%
	L2	h_{hand}	0.000100	4.803	1.431%
	L2	$h_{general}$	0.000500	4.806	1.365%
	L2	H_*	0.000500	4.807	1.349%
	L2	h_{hand}	0.005000	4.809	1.312%
	L2	h_{hand}	0.000050	4.812	1.245%
	L2	$h_{general}$	0.000100	4.862	0.215%
	L2	H_*	0.050000	4.864	0.177%
	L2	$h_{general}$	0.050000	4.866	0.137%
	L2	h_{hand}	0.050000	4.869	0.085%
	L2	H_*	0.001000	4.870	0.067%
	L2	$h_{general}$	0.010000	4.871	0.038%
	L2	h_{hand}	0.000500	4.878	-0.100%
	L2	H_*	0.000050	4.878	-0.114%
	L2	$h_{general}$	0.000050	4.882	-0.197%
	L2	H_*	0.005000	4.884	-0.230%
	L1	H_*	-	4.886	-0.264%
	L2	H_*	0.010000	4.903	-0.618%
	L2	$h_{general}$	0.001000	4.907	-0.690%
	L1	h_{hand}	-	4.914	-0.853%
	L1	CMoE	-	4.917	-0.913%
qr-decomposition		Baseline		1.673	0.000%
	L2	h_{hand}	0.001000	1.088	34.960%
	L2	$h_{general}$	0.000500	1.094	34.630%
	L2	CMoE	-	1.125	32.756%
	L2	$h_{general}$	0.000100	1.125	32.756%
	L2	$h_{general}$	0.010000	1.142	31.740%
	L1	h_{hand}	-	1.147	31.449%
	L2	$h_{general}$	0.000050	1.190	28.865%
	L2	$h_{general}$	0.050000	1.622	3.096%
	L1	H_*	-	1.622	3.044%
	L2	H_*	0.001000	1.623	3.008%
	L1	$h_{general}$	-	1.626	2.839%
	L2	$h_{general}$	0.005000	1.627	2.802%
	L2	h_{hand}	0.005000	1.628	2.730%
	L2	H_*	0.005000	1.629	2.673%
	L2	H_*	0.000050	1.632	2.455%
	L2	h_{hand}	0.050000	1.673	0.052%
	L1	CMoE	-	1.676	-0.135%
	L2	H_*	0.050000	1.680	-0.382%
	L2	h_{hand}	0.000050	1.724	-3.024%
	L2	h_{hand}	0.010000	1.757	-5.016%
	L2	H_*	0.000500	1.760	-5.178%
	L2	H_*	0.000100	1.762	-5.294%
	L2	H_*	0.010000	1.762	-5.311%
	L2	h_{hand}	0.000500	1.770	-5.793%
	L2	$h_{general}$	0.001000	1.856	-10.928%
	L2	h_{hand}	0.000100	2.067	-23.517%
sauvola		Baseline		6.706	0.000%
	L2	$h_{general}$	0.000500	6.620	1.292%
	L2	h_{hand}	0.010000	6.621	1.271%

	L1	h_{hand}	-	6.625	1.214%
	L2	$h_{general}$	0.000050	6.634	1.086%
	L2	$h_{general}$	0.010000	6.635	1.059%
	L2	h_{hand}	0.000500	6.638	1.024%
	L2	h_{hand}	0.005000	6.639	1.010%
	L2	h_{hand}	0.000100	6.641	0.973%
	L1	H_*	-	6.642	0.959%
	L2	H_*	0.000050	6.644	0.936%
	L1	CMoE	-	6.649	0.860%
	L2	h_{hand}	0.050000	6.652	0.812%
	L2	$h_{general}$	0.001000	6.654	0.781%
	L2	H_*	0.010000	6.655	0.773%
	L2	CMoE	-	6.658	0.718%
	L2	h_{hand}	0.000050	6.665	0.619%
	L2	H_*	0.005000	6.669	0.553%
	L2	$h_{general}$	0.000100	6.673	0.495%
	L1	$h_{general}$	-	6.675	0.464%
	L2	H_*	0.000500	6.676	0.457%
	L2	H_*	0.050000	6.680	0.398%
	L2	h_{hand}	0.001000	6.682	0.370%
	L2	H_*	0.001000	6.682	0.368%
	L2	$h_{general}$	0.050000	6.684	0.340%
	L2	$h_{general}$	0.005000	6.697	0.139%
	L2	H_*	0.000100	6.715	-0.133%
chameneos-redux-evtchn		Baseline		4.787	0.000%
	L2	H_*	0.001000	4.668	2.480%
	L1	H_*	-	4.694	1.937%
	L2	CMoE	-	4.743	0.912%
	L2	h_{hand}	0.000100	4.744	0.896%
	L2	h_{hand}	0.005000	4.752	0.736%
	L2	$h_{general}$	0.005000	4.754	0.686%
	L2	h_{hand}	0.000500	4.765	0.468%
	L1	h_{hand}	-	4.767	0.417%
	L2	H_*	0.010000	4.767	0.415%
	L2	$h_{general}$	0.001000	4.776	0.232%
	L2	H_*	0.050000	4.776	0.231%
	L1	$h_{general}$	-	4.784	0.052%
	L2	h_{hand}	0.010000	4.787	0.002%
	L2	$h_{general}$	0.050000	4.791	-0.087%
	L2	H_*	0.000100	4.794	-0.154%
	L2	h_{hand}	0.001000	4.797	-0.214%
	L2	H_*	0.000500	4.800	-0.279%
	L2	h_{hand}	0.000050	4.803	-0.330%
	L2	H_*	0.000050	4.808	-0.430%
	L2	$h_{general}$	0.000500	4.813	-0.538%
	L2	$h_{general}$	0.010000	4.818	-0.657%
	L2	$h_{general}$	0.000100	4.819	-0.659%
	L2	H_*	0.005000	4.831	-0.927%
	L2	h_{hand}	0.050000	4.846	-1.226%
	L1	CMoE	-	4.853	-1.376%
	L2	$h_{general}$	0.000050	4.867	-1.673%
valet-async		Baseline		5.912	0.000%
	L2	$h_{general}$	0.000050	5.741	2.889%
	L2	h_{hand}	0.000050	5.758	2.597%
	L2	$h_{general}$	0.000500	5.760	2.561%
	L2	h_{hand}	0.000100	5.769	2.412%
	L2	$h_{general}$	0.005000	5.796	1.966%
	L2	h_{hand}	0.000500	5.853	1.001%

	L2	$h_{general}$	0.000100	5.887	0.429%
	L2	H_*	0.000050	5.890	0.364%
	L1	$h_{general}$	-	5.901	0.188%
	L2	H_*	0.000100	5.913	-0.021%
	L2	$h_{general}$	0.050000	5.914	-0.043%
	L2	$h_{general}$	0.001000	5.917	-0.085%
	L2	H_*	0.010000	5.923	-0.182%
	L2	H_*	0.001000	5.938	-0.445%
	L2	h_{hand}	0.010000	5.950	-0.642%
	L1	H_*	-	6.171	-4.387%
	L2	CMoE	-	6.189	-4.681%
	L2	h_{hand}	0.005000	6.232	-5.418%
	L2	h_{hand}	0.001000	6.254	-5.786%
	L2	$h_{general}$	0.010000	6.369	-7.725%
	L1	CMoE	-	6.372	-7.779%
	L2	h_{hand}	0.050000	6.380	-7.920%
	L2	H_*	0.050000	6.385	-8.009%
	L1	h_{hand}	-	6.392	-8.118%
	L2	H_*	0.000500	6.395	-8.174%
	L2	H_*	0.005000	6.402	-8.288%
lu-decomposition		Baseline		6.471	0.000%
	L2	$h_{general}$	0.000500	6.405	1.014%
	L2	H_*	0.010000	6.406	1.002%
	L2	$h_{general}$	0.000050	6.407	0.994%
	L1	CMoE	-	6.426	0.694%
	L2	H_*	0.005000	6.431	0.611%
	L1	h_{hand}	-	6.433	0.595%
	L2	h_{hand}	0.001000	6.436	0.543%
	L2	$h_{general}$	0.050000	6.437	0.527%
	L1	$h_{general}$	-	6.438	0.514%
	L2	H_*	0.050000	6.439	0.491%
	L2	H_*	0.001000	6.441	0.471%
	L2	h_{hand}	0.050000	6.461	0.156%
	L2	h_{hand}	0.010000	6.462	0.132%
	L2	$h_{general}$	0.010000	6.467	0.069%
	L2	h_{hand}	0.005000	6.474	-0.042%
	L2	CMoE	-	6.486	-0.239%
	L2	$h_{general}$	0.001000	6.512	-0.632%
	L2	$h_{general}$	0.000100	6.517	-0.718%
	L2	h_{hand}	0.000050	6.543	-1.115%
	L2	h_{hand}	0.000500	6.545	-1.142%
	L1	H_*	-	6.553	-1.261%
	L2	$h_{general}$	0.005000	6.564	-1.444%
	L2	h_{hand}	0.000100	6.845	-5.786%
	L2	H_*	0.000500	6.945	-7.329%
	L2	H_*	0.000050	6.949	-7.388%
	L2	H_*	0.000100	6.954	-7.465%
lens		Baseline		9.691	0.000%
	L2	h_{hand}	0.000050	4.450	54.083%
	L2	$h_{general}$	0.000100	4.450	54.083%
	L2	$h_{general}$	0.000050	4.451	54.073%
	L2	H_*	0.000050	4.451	54.072%
	L2	$h_{general}$	0.050000	4.455	54.031%
	L1	H_*	-	4.492	53.646%
	L2	h_{hand}	0.000100	4.492	53.646%
	L2	H_*	0.000100	4.492	53.646%
	L1	$h_{general}$	-	4.492	53.645%
	L2	H_*	0.000500	4.492	53.644%

	L2	$h_{general}$	0.000500	4.492	53.644%
	L2	h_{hand}	0.000500	4.493	53.638%
	L2	CMoE	-	4.494	53.631%
	L2	$h_{general}$	0.001000	4.496	53.601%
	L1	CMoE	-	4.500	53.564%
	L2	h_{hand}	0.050000	9.689	0.021%
	L1	h_{hand}	-	9.690	0.012%
	L2	$h_{general}$	0.010000	9.690	0.012%
	L2	$h_{general}$	0.005000	9.691	0.001%
	L2	H_*	0.005000	9.763	-0.749%
	L2	H_*	0.001000	9.764	-0.753%
	L2	H_*	0.050000	9.766	-0.776%
	L2	H_*	0.010000	9.767	-0.784%
	L2	h_{hand}	0.005000	9.805	-1.176%
	L2	h_{hand}	0.001000	9.942	-2.595%
	L2	h_{hand}	0.010000	9.948	-2.652%
fyq-symbolic-maths		Baseline		1.184	0.000%
	L2	$h_{general}$	0.000500	1.175	0.744%
	L2	H_*	0.050000	1.184	0.006%
	L2	$h_{general}$	0.005000	1.184	-0.011%
	L2	$h_{general}$	0.050000	1.184	-0.023%
	L2	h_{hand}	0.050000	1.184	-0.023%
	L2	H_*	0.000100	1.184	-0.028%
	L1	CMoE	-	1.184	-0.034%
	L2	H_*	0.005000	1.184	-0.034%
	L2	$h_{general}$	0.010000	1.184	-0.045%
	L2	H_*	0.000500	1.184	-0.051%
	L2	$h_{general}$	0.000100	1.184	-0.051%
	L2	h_{hand}	0.000500	1.184	-0.073%
	L2	h_{hand}	0.001000	1.184	-0.073%
	L2	H_*	0.000050	1.185	-0.079%
	L2	H_*	0.001000	1.185	-0.079%
	L2	$h_{general}$	0.001000	1.185	-0.101%
	L2	$h_{general}$	0.000050	1.185	-0.107%
	L2	h_{hand}	0.005000	1.185	-0.112%
	L1	H_*	-	1.185	-0.146%
	L1	h_{hand}	-	1.186	-0.175%
	L2	CMoE	-	1.188	-0.347%
	L2	h_{hand}	0.000100	1.188	-0.382%
	L2	H_*	0.010000	1.189	-0.444%
	L2	h_{hand}	0.010000	1.189	-0.450%
	L2	h_{hand}	0.000050	1.190	-0.551%
	L1	$h_{general}$	-	1.191	-0.603%
sieve		Baseline		7.897	0.000%
	L2	CMoE	-	6.460	18.194%
	L2	H_*	0.010000	6.485	17.869%
	L2	h_{hand}	0.000500	6.523	17.397%
	L2	H_*	0.005000	6.564	16.875%
	L1	$h_{general}$	-	6.580	16.676%
	L2	h_{hand}	0.010000	6.584	16.621%
	L2	H_*	0.000100	6.708	15.050%
	L2	h_{hand}	0.005000	6.779	14.152%
	L2	$h_{general}$	0.000050	6.848	13.281%
	L2	h_{hand}	0.000050	7.237	8.354%
	L2	H_*	0.000050	7.329	7.189%
	L2	H_*	0.000500	7.650	3.119%
	L2	$h_{general}$	0.001000	7.864	0.413%
	L2	H_*	0.001000	7.866	0.388%

	L2	$h_{general}$	0.050000	7.898	-0.018%
	L2	H_*	0.050000	7.900	-0.041%
	L2	h_{hand}	0.001000	7.900	-0.050%
	L2	h_{hand}	0.000100	7.901	-0.062%
	L2	h_{hand}	0.050000	7.903	-0.087%
	L2	$h_{general}$	0.010000	7.905	-0.106%
	L2	$h_{general}$	0.000500	7.968	-0.899%
	L1	H_*	-	7.975	-0.995%
	L1	CMoE	-	7.978	-1.036%
	L1	h_{hand}	-	7.995	-1.247%
	L2	$h_{general}$	0.005000	8.001	-1.318%
	L2	$h_{general}$	0.000100	8.008	-1.408%
fyq-stdlib-functor-record-sets		Baseline		7.011	0.000%
	L2	H_*	0.000500	6.882	1.836%
	L2	$h_{general}$	0.000100	6.982	0.402%
	L2	h_{hand}	0.000050	6.998	0.186%
	L2	CMoE	-	6.998	0.184%
	L2	h_{hand}	0.000100	7.006	0.064%
	L2	h_{hand}	0.005000	7.008	0.040%
	L2	h_{hand}	0.000500	7.008	0.036%
	L2	H_*	0.000100	7.009	0.021%
	L1	CMoE	-	7.011	-0.003%
	L2	h_{hand}	0.001000	7.011	-0.003%
	L2	$h_{general}$	0.001000	7.012	-0.016%
	L1	H_*	-	7.012	-0.019%
	L2	$h_{general}$	0.000050	7.013	-0.032%
	L2	$h_{general}$	0.050000	7.013	-0.038%
	L2	H_*	0.050000	7.015	-0.058%
	L2	$h_{general}$	0.000500	7.015	-0.064%
	L2	H_*	0.000050	7.016	-0.080%
	L2	H_*	0.005000	7.018	-0.108%
	L2	H_*	0.001000	7.018	-0.108%
	L1	$h_{general}$	-	7.043	-0.465%
	L2	H_*	0.010000	7.047	-0.520%
	L1	h_{hand}	-	7.052	-0.589%
	L2	h_{hand}	0.010000	7.065	-0.773%
	L2	$h_{general}$	0.010000	7.075	-0.916%
	L2	h_{hand}	0.050000	7.092	-1.154%
	L2	$h_{general}$	0.005000	7.095	-1.199%
kb		Baseline		8.426	0.000%
	L2	h_{hand}	0.001000	7.697	8.652%
	L2	$h_{general}$	0.000050	7.739	8.144%
	L2	$h_{general}$	0.000500	7.806	7.352%
	L2	H_*	0.001000	7.814	7.257%
	L2	H_*	0.000050	7.872	6.574%
	L2	$h_{general}$	0.001000	7.919	6.009%
	L2	h_{hand}	0.005000	7.926	5.930%
	L2	H_*	0.000100	7.971	5.398%
	L2	h_{hand}	0.010000	8.053	4.425%
	L1	$h_{general}$	-	8.090	3.982%
	L1	H_*	-	8.098	3.893%
	L2	CMoE	-	8.136	3.439%
	L2	H_*	0.000500	8.140	3.383%
	L2	$h_{general}$	0.000100	8.252	2.061%
	L2	h_{hand}	0.000100	8.273	1.813%
	L2	h_{hand}	0.000500	8.316	1.304%
	L2	$h_{general}$	0.010000	8.343	0.983%
	L2	h_{hand}	0.050000	8.348	0.915%

	L1	CMoE	-	8.374	0.614%
	L2	H_*	0.005000	8.395	0.364%
	L2	H_*	0.050000	8.400	0.298%
	L2	$h_{general}$	0.050000	8.406	0.231%
	L1	h_{hand}	-	8.452	-0.314%
	L2	$h_{general}$	0.005000	8.457	-0.371%
	L2	h_{hand}	0.000050	8.470	-0.526%
	L2	H_*	0.010000	8.557	-1.555%
fyq-stdlib-int-sets		Baseline		3.923	0.000%
	L1	CMoE	-	3.883	1.040%
	L1	h_{hand}	-	3.893	0.777%
	L2	$h_{general}$	0.001000	3.893	0.775%
	L2	h_{hand}	0.005000	3.894	0.743%
	L2	H_*	0.000100	3.896	0.709%
	L2	h_{hand}	0.000100	3.896	0.698%
	L2	H_*	0.010000	3.896	0.692%
	L2	h_{hand}	0.000500	3.896	0.686%
	L2	h_{hand}	0.000050	3.897	0.678%
	L2	h_{hand}	0.001000	3.897	0.678%
	L2	$h_{general}$	0.000500	3.897	0.671%
	L2	H_*	0.000500	3.898	0.646%
	L2	H_*	0.000050	3.898	0.642%
	L2	$h_{general}$	0.000050	3.899	0.632%
	L2	H_*	0.050000	3.901	0.562%
	L1	H_*	-	3.905	0.457%
	L2	H_*	0.005000	3.907	0.416%
	L1	$h_{general}$	-	3.912	0.296%
	L2	h_{hand}	0.050000	3.920	0.090%
	L2	$h_{general}$	0.050000	3.921	0.058%
	L2	H_*	0.001000	3.956	-0.843%
	L2	$h_{general}$	0.000100	3.958	-0.880%
	L2	$h_{general}$	0.010000	3.959	-0.907%
	L2	$h_{general}$	0.005000	3.962	-0.984%
	L2	CMoE	-	3.985	-1.560%
	L2	h_{hand}	0.010000	3.989	-1.681%
format-benchmark		Baseline		7.212	0.000%
	L2	H_*	0.000050	6.715	6.902%
	L2	H_*	0.000100	6.717	6.875%
	L2	$h_{general}$	0.000050	6.754	6.351%
	L2	$h_{general}$	0.000100	6.770	6.140%
	L2	h_{hand}	0.001000	6.863	4.844%
	L1	H_*	-	6.908	4.219%
	L2	h_{hand}	0.000500	6.915	4.120%
	L2	H_*	0.001000	6.956	3.554%
	L2	H_*	0.000500	7.028	2.554%
	L2	h_{hand}	0.000050	7.029	2.538%
	L2	h_{hand}	0.000100	7.048	2.280%
	L2	$h_{general}$	0.000500	7.080	1.838%
	L1	h_{hand}	-	7.104	1.499%
	L2	CMoE	-	7.108	1.450%
	L2	H_*	0.005000	7.111	1.407%
	L2	H_*	0.010000	7.130	1.137%
	L2	h_{hand}	0.005000	7.137	1.046%
	L1	CMoE	-	7.141	0.991%
	L1	$h_{general}$	-	7.142	0.981%
	L2	$h_{general}$	0.005000	7.148	0.897%
	L2	$h_{general}$	0.001000	7.175	0.515%
	L2	H_*	0.050000	7.181	0.436%

	L2	h_{hand}	0.010000	7.184	0.389%
	L2	$h_{general}$	0.050000	7.185	0.379%
	L2	$h_{general}$	0.010000	7.216	-0.050%
	L2	h_{hand}	0.050000	7.275	-0.865%
hamming		Baseline		12.611	0.000%
	L1	H_*	-	12.067	4.314%
	L2	H_*	0.001000	12.202	3.243%
	L1	$h_{general}$	-	12.242	2.930%
	L2	h_{hand}	0.005000	12.276	2.657%
	L2	H_*	0.000100	12.286	2.578%
	L2	$h_{general}$	0.000500	12.318	2.321%
	L2	h_{hand}	0.000500	12.328	2.244%
	L2	$h_{general}$	0.001000	12.330	2.228%
	L2	H_*	0.005000	12.389	1.758%
	L1	h_{hand}	-	12.398	1.692%
	L2	h_{hand}	0.010000	12.402	1.657%
	L2	$h_{general}$	0.010000	12.403	1.647%
	L2	H_*	0.010000	12.412	1.581%
	L2	H_*	0.000050	12.430	1.438%
	L2	h_{hand}	0.000100	12.437	1.378%
	L2	H_*	0.000500	12.453	1.253%
	L2	$h_{general}$	0.000100	12.473	1.096%
	L2	h_{hand}	0.050000	12.476	1.067%
	L2	h_{hand}	0.000050	12.497	0.908%
	L2	CMoE	-	12.511	0.790%
	L2	$h_{general}$	0.005000	12.516	0.753%
	L2	h_{hand}	0.001000	12.524	0.691%
	L1	CMoE	-	12.541	0.553%
	L2	$h_{general}$	0.000050	12.564	0.375%
	L2	$h_{general}$	0.050000	12.603	0.064%
	L2	H_*	0.050000	12.715	-0.821%
chameneos-redux-th		Baseline		2.381	0.000%
	L2	H_*	0.000500	2.276	4.409%
	L2	h_{hand}	0.010000	2.325	2.368%
	L2	$h_{general}$	0.001000	2.326	2.324%
	L2	h_{hand}	0.000500	2.329	2.179%
	L2	H_*	0.050000	2.329	2.167%
	L2	CMoE	-	2.331	2.102%
	L1	$h_{general}$	-	2.333	2.025%
	L2	h_{hand}	0.050000	2.335	1.909%
	L2	$h_{general}$	0.000050	2.337	1.830%
	L1	h_{hand}	-	2.339	1.764%
	L2	$h_{general}$	0.000500	2.339	1.762%
	L1	H_*	-	2.339	1.741%
	L2	h_{hand}	0.000050	2.340	1.730%
	L2	h_{hand}	0.005000	2.341	1.658%
	L1	CMoE	-	2.343	1.584%
	L2	H_*	0.001000	2.345	1.501%
	L2	$h_{general}$	0.000100	2.348	1.384%
	L2	h_{hand}	0.000100	2.350	1.297%
	L2	$h_{general}$	0.005000	2.359	0.903%
	L2	$h_{general}$	0.050000	2.362	0.794%
	L2	H_*	0.005000	2.362	0.790%
	L2	h_{hand}	0.001000	2.366	0.608%
	L2	H_*	0.000050	2.367	0.582%
	L2	$h_{general}$	0.010000	2.368	0.529%
	L2	H_*	0.000100	2.369	0.516%
	L2	H_*	0.010000	2.390	-0.395%

boyer		Baseline		5.820	0.000%
	L2	H_*	0.005000	5.408	7.074%
	L1	$h_{general}$	-	5.408	7.072%
	L2	$h_{general}$	0.000500	5.483	5.789%
	L2	h_{hand}	0.000500	5.486	5.732%
	L2	CMoE	-	5.510	5.332%
	L2	$h_{general}$	0.001000	5.525	5.072%
	L2	$h_{general}$	0.000050	5.542	4.766%
	L2	$h_{general}$	0.000100	5.545	4.725%
	L2	$h_{general}$	0.005000	5.547	4.685%
	L2	h_{hand}	0.000100	5.564	4.396%
	L2	H_*	0.000500	5.701	2.033%
	L2	h_{hand}	0.000050	5.708	1.928%
	L2	h_{hand}	0.001000	5.709	1.906%
	L2	$h_{general}$	0.010000	5.710	1.887%
	L2	H_*	0.000050	5.711	1.875%
	L2	H_*	0.001000	5.721	1.699%
	L2	H_*	0.000100	5.724	1.652%
	L1	H_*	-	5.733	1.497%
	L2	H_*	0.010000	5.743	1.315%
	L2	$h_{general}$	0.050000	5.780	0.683%
	L2	h_{hand}	0.010000	5.816	0.062%
	L2	H_*	0.050000	5.818	0.023%
	L2	h_{hand}	0.050000	5.842	-0.379%
	L1	CMoE	-	5.863	-0.735%
	L1	h_{hand}	-	5.866	-0.790%
	L2	h_{hand}	0.005000	5.869	-0.849%

Appendix F

Benchmarks

The source code for the benchmarks used in the project can be acceessed at https://github.com/fyquah95/fyp-experiments.

F.1 Training Benchmarks

`almabench`	*normal/almabench*
`bdd`	*normal/bdd_benchmark*
`fft`	*normal/fft*
`floats-in-functor`	*normal/floats-in-functor*
`hamming`	*normal/hamming*
`kahan-sum`	*normal/kahan_sum*
`kb-benchmark`	*normal/kb_benchmark*
`lens-benchmark`	*normal/lens_benchmark*
`lexifi`	*normal/lexifi-g2pp_benchmark*
`quicksort`	*normal/quicksort*
`sequence`	*normal/sequence_benchmark*
`sequence-cps`	*normal/sequence_cps_benchmark*

F.2 Testing Benchmarks

`boyer`	*micro-evaluation/boyer*
`chameneos-redux-async`	*micro-evaluation/chameneos-redux-async*
`chameneos-redux-evtchn`	*micro-evaluation/chameneos-redux-evtchn*
`chameneos-redux-lwt`	*micro-evaluation/chameneos-redux-lwt*
`chameneos-redux-th`	*micro-evaluation/chameneos-redux-th*
`format-benchmark`	*micro-evaluation/format-benchmark*
`list-benchmark`	*micro-evaluation/list-benchmark*
`nucleic`	*micro-evaluation/nucleic*
`nullable-array`	*micro-evaluation/nullable-array*
`sauvola`	*micro-evaluation/sauvola*
`sieve`	*micro-evaluation/sieve*
`valet-async`	*micro-evaluation/valet-async*
`valet-lwt`	*micro-evaluation/valet-lwt*
`vector-functor`	*micro-evaluation/vector-functor*
`levinson-durbin`	*micro-evaluation/numerical/levinson-durbin*
`lu-decomposition`	*micro-evaluation/numerical/lu-decomposition*
`naive-multilayer`	*micro-evaluation/numerical/naive-multilayer*
`qr-decomposition`	*micro-evaluation/numerical/qr-decomposition*

F.3 Pathelogical Benchmarks

These benchmark are curated to test edge cases in making inlining decisions.

`rev-list`	*fyq/rev-list*
`symbolic-maths`	*fyq/symbolic-maths*
`stdlib-functor-record-sets`	*fyq/stdlib-functor-record-sets*
`stdlib-int-sets`	*fyq/stdlib-int-sets*

Bibliography

Aarts, E. & Korst, J. (1988), 'Simulated annealing and boltzmann machines'.

Agakov, F., Bonilla, E., Cavazos, J., Franke, B., Fursin, G., O'Boyle, M. F., Thomson, J., Toussaint, M. & Williams, C. K. (2006), Using machine learning to focus iterative optimization, *in* 'Proceedings of the International Symposium on Code Generation and Optimization', IEEE Computer Society, pp. 295–305.

Arthur, D. & Vassilvitskii, S. (2007), k-means++: The advantages of careful seeding, *in* 'Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms', Society for Industrial and Applied Mathematics, pp. 1027–1035.

Augustsson, L., Rittri, M. & Synek, D. (1994), 'Functional pearl: On generating unique names', *Journal of Functional Programming* **4**(1), 117–123.

Avnimelech, R. & Intrator, N. (1999), 'Boosted mixture of experts: an ensemble learning scheme', *Neural computation* **11**(2), 483–497.

Bacon, D. F., Graham, S. L. & Sharp, O. J. (1994), 'Compiler transformations for high-performance computing', *ACM Computing Surveys (CSUR)* **26**(4), 345–420.

Bailey, K. (1994), 'Numerical taxonomy and cluster analysis', *Typologies and Taxonomies* **34**, 24.

Beyer, K., Goldstein, J., Ramakrishnan, R. & Shaft, U. (1999), When is "nearest neighbor" meaningful?, *in* 'In Int. Conf. on Database Theory', pp. 217–235.

Bottou, L. (2010), Large-scale machine learning with stochastic gradient descent, *in* 'Proceedings of COMPSTAT'2010', Springer, pp. 177–186.

Cavazos, J. & O'Boyle, M. F. (2005), Automatic tuning of inlining heuristics, *in* 'Proceedings of the 2005 ACM/IEEE conference on Supercomputing', IEEE Computer Society, p. 14.

Chambart, P., Shinwell, M. & White, L. (2017), 'Ocaml – middle_end/inlining_cost.ml'.
URL: *https://github.com/ocaml/ocaml*

Cheney, J. & Hinze, R. (2003), Phantom types, Technical report, Citeseer.

Christiano, P. F., Leike, J., Brown, T., Martic, M., Legg, S. & Amodei, D. (2017), Deep reinforcement learning from human preferences, *in* 'Advances in Neural Information Processing Systems', pp. 4302–4310.

Cocke, J. (1970), 'Programming languages and their compilers: Preliminary notes'.

contributors, W. (2017), 'Explicit data graph execution — wikipedia, the free encyclopedia', https://en.wikipedia.org/w/index.php?title=Explicit_data_graph_execution&oldid=761541137. [Online; accessed 29-January-2018].

Coons, K. E., Robatmili, B., Taylor, M. E., Maher, B. A., Burger, D. & McKinley, K. S. (2008), Feature selection and policy optimization for distributed instruction placement using reinforcement learning, *in* 'Proceedings of the 17th international conference on Parallel architectures and compilation techniques', ACM, pp. 32–42.

Cooper, K. D., Hall, M. W. & Torczon, L. (1992), 'Unexpected side effects of inline substitution: A case study', *ACM Letters on Programming Languages and Systems (LOPLAS)* **1**(1), 22–32.

Coutts, D., Leshchinskiy, R. & Stewart, D. (2007), Stream fusion: From lists to streams to nothing at all, *in* 'ACM SIGPLAN Notices', Vol. 42, ACM, pp. 315–326.

Cummins, C., Petoumenos, P., Wang, Z. & Leather, H. (2017*a*), End-to-end deep learning of optimization heuristics, *in* 'Parallel Architectures and Compilation Techniques (PACT), 2017 26th International Conference on', IEEE, pp. 219–232.

Cummins, C., Petoumenos, P., Wang, Z. & Leather, H. (2017*b*), Synthesizing benchmarks for predictive modeling, *in* 'Code Generation and Optimization (CGO), 2017 IEEE/ACM International Symposium on', IEEE, pp. 86–99.

Damas, L. & Milner, R. (1982), Principal type-schemes for functional programs, *in* 'Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages', ACM, pp. 207–212.

De La Briandais, R. (1959), File searching using variable length keys, *in* 'Papers presented at the the March 3-5, 1959, western joint computer conference', ACM, pp. 295–298.

Fernandéz, M. (2009), 'When immutable data is faster: write barrier costs'.
URL: *http://web.archive.org/web/20100816130113/http://eigenclass.org/R2/writings/write-barrier-cost*

Fisher, R. A. (1936), 'The use of multiple measurements in taxonomic problems', *Annals of human genetics* **7**(2), 179–188.

Gers, F. A., Schmidhuber, J. & Cummins, F. (1999), 'Learning to forget: Continual prediction with lstm'.

GHC (2015), 10.31.6.1. inline pragma, *in* 'Glasgow Haskell Compiler User's Guide', chapter 10.31.6.1.

Greene, W. H. (2003), *Econometric analysis*, Pearson Education India.

Hall, C. V., Hammond, K., Peyton Jones, S. L. & Wadler, P. L. (1996), 'Type classes in haskell', *ACM Transactions on Programming Languages and Systems (TOPLAS)* **18**(2), 109–138.

Hamerly, G. & Elkan, C. (2002), Alternatives to the k-means algorithm that find better clusterings, *in* 'Proceedings of the eleventh international conference on Information and knowledge management', ACM, pp. 600–607.

Hartigan, J. A. & Wong, M. A. (1979), 'Algorithm as 136: A k-means clustering algorithm', *Journal of the Royal Statistical Society. Series C (Applied Statistics)* **28**(1), 100–108.

Hotelling, H. (1933), 'Analysis of a complex of statistical variables into principal components.', *Journal of educational psychology* **24**(6), 417.

Hotelling, H. (1936), 'Relations between two sets of variates', *Biometrika* **28**(3/4), 321–377.

Hudak, P., Wadler, P. et al. (1989), 'Report on the functional programming language haskell, version 1.0 pre-release', *Distributed at FPCA* **89**.

Inc, R. (2018), 'Redhat enterprise linux : 6.3 configuration suggestions'.
URL: *https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/performance_tuning_guide/sect-red_hat_enterprise_linux-performance_tuning_guide-cpu-configuration_suggestions*

Jacobs, R. A., Jordan, M. I., Nowlan, S. J. & Hinton, G. E. (1991), 'Adaptive mixtures of local experts', *Neural computation* **3**(1), 79–87.

Jagannathan, S. & Wright, A. (1996), Flow-directed inlining, *in* 'ACM SIGPLAN Notices', Vol. 31, ACM, pp. 193–205.

Jones, S. P., Hall, C., Hammond, K., Partain, W. & Wadler, P. (1993), The glasgow haskell compiler: a technical overview, *in* 'Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference', Vol. 93.

Jones, S. P. & Marlow, S. (2002), 'Secrets of the glasgow haskell compiler inliner', *Journal of Functional Programming* **12**(4-5), 393–434.

Kaelbling, L. P., Littman, M. L. & Moore, A. W. (1996), 'Reinforcement learning: A survey', *Journal of artificial intelligence research* **4**, 237–285.

Kahneman, D. & Egan, P. (2011), *Thinking, fast and slow*, Vol. 1, Farrar, Straus and Giroux New York.

Kaser, O. & Ramakrishnan, C. (1998), 'Evaluating inlining techniques', *Computer Languages* **24**(2), 55–72.

Knijnenburg, P. M. W., Kisuki, T. & O'Boyle, M. F. P. (2002), Embedded processor design challenges, Springer-Verlag New York, Inc., New York, NY, USA, chapter Iterative Compilation, pp. 171–187.
URL: *http://dl.acm.org/citation.cfm?id=765198.765209*

Knuth, D. E. & Moore, R. W. (1975), 'An analysis of alpha-beta pruning', *Artificial intelligence* **6**(4), 293–326.

Kock, A. (1970), 'Monads on symmetric monoidal closed categories', *Archiv der Mathematik* **21**(1), 1–10.

Kocsis, L. & Szepesvári, C. (2006), Bandit based monte-carlo planning, *in* 'ECML', Vol. 6, Springer, pp. 282–293.

KPFRS, L. (1901), On lines and planes of closest fit to systems of points in space, *in* 'Proceedings of the 17th ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems (SIGMOD)'.

Kulkarni, S., Cavazos, J., Wimmer, C. & Simon, D. (2013), Automatic construction of inlining heuristics using machine learning, *in* 'Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)', pp. 1–12.

Lattner, C. & Adve, V. (2004), Llvm: A compilation framework for lifelong program analysis & transformation, *in* 'Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization', IEEE Computer Society, p. 75.

Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Remy, D. & Vouillon, J. (2017), 'The ocaml system release 4.06 documentation and user's manual'.
URL: *http://caml.inria.fr/pub/docs/manual-ocaml/*

LLC, J. S. & Contributors (2018*a*), 'Async library'.
URL: *https://github.com/janestreet/async*

LLC, J. S. & Contributors (2018*b*), 'Core standard library'.
URL: *https://github.com/janestreet/core*

Lopes, N. P. (2010), 'Boosting inlining heuristics'.

MacKay, D. J. & Mac Kay, D. J. (2003), *Information theory, inference and learning algorithms*, Cambridge university press.

Magalhaes, J. P., Holdermans, S., Jeuring, J. & Löh, A. (2010), Optimizing generics is easy!, *in* 'Proceedings of the 2010 ACM SIGPLAN workshop on Partial evaluation and program manipulation', ACM, pp. 33–42.

Mainland, G., Leshchinskiy, R., Jones, S. P. & Marlow, S. (2013), 'Haskell beats c using generalized stream fusion', *Under submission* .

McCulloch, W. S. & Pitts, W. (1943), 'A logical calculus of the ideas immanent in nervous activity', *The bulletin of mathematical biophysics* **5**(4), 115–133.

Milner, R. (1997), *The definition of standard ML: revised*, MIT press.

Minsky, Y., Madhavapeddy, A. & Hickey, J. (2013*a*), Memory representation of values, *in* 'Real World OCaml: Functional programming for the masses', " O'Reilly Media, Inc.", chapter 20.

Minsky, Y., Madhavapeddy, A. & Hickey, J. (2013*b*), Understanding the garbage collector, *in* 'Real World OCaml: Functional programming for the masses', " O'Reilly Media, Inc.", chapter 21.

Nethercote, N. & Seward, J. (2003), 'Valgrind: A program supervision framework', *Electronic notes in theoretical computer science* **89**(2), 44–66.

Ng, A. Y. (2004), Feature selection, l 1 vs. l 2 regularization, and rotational invariance, *in* 'Proceedings of the twenty-first international conference on Machine learning', ACM, p. 78.

Ng, A. Y., Russell, S. J. et al. (2000), Algorithms for inverse reinforcement learning., *in* 'Icml', pp. 663–670.

Peterson, J., Hammond, K., Augustsson, L., Boutel, B., Burton, F., Fasel, J., Gordon, A., Hughes, R., Hudak, P., Johnsson, T. et al. (1996), 'Report on the functional programming language haskell, version 1.3', *preperation Yale University and University of St Andrews* .

Quinlan, J. R. (1986), 'Induction of decision trees', *Machine learning* **1**(1), 81–106.

Rao, C. R. (1948), 'The utilization of multiple measurements in problems of biological classification', *Journal of the Royal Statistical Society. Series B (Methodological)* **10**(2), 159–203.

Rumelhart, D. E., Hinton, G. E. & Williams, R. J. (1986), 'Learning representations by back-propagating errors', *nature* **323**(6088), 533.

Russell, S. (1998), Learning agents for uncertain environments, *in* 'Proceedings of the eleventh annual conference on Computational learning theory', ACM, pp. 101–103.

Sabry, A. & Felleisen, M. (1992), *Reasoning about programs in continuation-passing style.*, number 1, ACM.

Samuel, A. L. (1959), 'Some studies in machine learning using the game of checkers', *IBM Journal of research and development* **3**(3), 210–229.

Scheifler, R. W. (1977), 'An analysis of inline substitution for a structured programming language', *Communications of the ACM* **20**(9), 647–654.

Schkufza, E., Sharma, R. & Aiken, A. (2013), Stochastic superoptimization, *in* 'ACM SIGARCH Computer Architecture News', Vol. 41, ACM, pp. 305–316.

Serrano, M. (1997), Inline expansion: when and how?, *in* 'International Symposium on Programming Language Implementation and Logic Programming', Springer, pp. 143–157.

Shivers, O. (1991), The semantics of scheme control-flow analysis, *in* 'Proceedings of the 1991 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation', PEPM '91, ACM, New York, NY, USA, pp. 190–198.
URL: *http://doi.acm.org/10.1145/115865.115884*

Stanley, K. O. & Miikkulainen, R. (2002), 'Evolving neural networks through augmenting topologies', *Evolutionary computation* **10**(2), 99–127.

Sutton, R. S. & Barto, A. G. (1998), *Reinforcement learning: An introduction*, Vol. 1, MIT press Cambridge.

Taleb, N. (2005), *Fooled by randomness: The hidden role of chance in life and in the markets*, Vol. 1, Random House Incorporated.

Tharwat, A., Gaber, T., Ibrahim, A. & Hassanien, A. E. (2017), 'Linear discriminant analysis: A detailed tutorial', **30**, 169–190,.

Tibshirani, R. (1996), 'Regression shrinkage and selection via the lasso', *Journal of the Royal Statistical Society. Series B (Methodological)* pp. 267–288.

Torvalds, L. (2001), 'Re: Sigh. inlining heuristics.'.
URL: *https://gcc.gnu.org/ml/gcc/2001-07/msg00916.html*

Walker, S. H. & Duncan, D. B. (1967), 'Estimation of the probability of an event as a function of several independent variables', *Biometrika* **54**(1-2), 167–179.

Wang, Z. & O'Boyle, M. (2018), 'Machine learning in compiler optimization', *Proceedings of the IEEE* .

White, L., Bour, F. & Yallop, J. (2015), 'Modular implicits', *arXiv preprint arXiv:1512.01895* .

Stable Abstract Syntax Tree Labeling

FU YONG QUAH, Imperial College London, United Kingdom
DAVID THOMAS, Imperial College London, United Kingdom

With the recent interest in self-optimising compilers, the usage of third-party optimisers in making optimisation decision in abstract syntax tree (AST) nodes to interact with industrial-grade compilers is of interest. This requires some reliable conventions for labeling nodes within an AST. An algorithm for consistently and deterministically labeling nodes in AST is hence presented. The labeling algorithm can reliably process $merge(tree, u, v)$ operation operations, whereby a single leaf node v is replaced withwith the transitive closure of u, resulting in a transformation of the AST anlogous to function inlining. An example of this operation is depicted in figure 1.

This labeling algorithm fulfils three important functional properties, namely (1) the generated node identifiers are reflects the absolute amount of transformation performed to arrive at the given tree structure. (2) locally unique identifiers (ie: the identifiers across sibling nodes are unique) (3) Node identifiers are stable, that is, they are independent of node identifiers of siblings and descendants. The algorithm supports ASTs with high-order functions, namely allowing nested function declarations, uses constant time and memory to generate a new identifier for nodes added to the tree as a result of *merge* operations, and hence does not contribute to significant overhead on top of other tree manipulation due to existing compiler passes.

CCS Concepts: • **Theory of computation** → *Data structures design and analysis*;

Additional Key Words and Phrases: Language Processors, Naming, Abstract Syntax Tree

ACM Reference Format:
Fu Yong Quah and David Thomas. 2018. Stable Abstract Syntax Tree Labeling. 1, 1 (May 2018), 9 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

Name generation for nodes in AST is commonly used in compilers and language processors. A common practice is to label nodes and generate variable names with ad-hoc unique identifiers, such as the `gensym` function as described by [1].

```
int gensym() {
  static int i = 0;
  return i++;
}
```

However, when there exists multiple instances of ASTs that have undergone a different set of manipulations, a stable labeling algorithm is important as it allows programs to identify and compare nodes that exist in both trees. A common manipulation is the merge operation,

Authors' addresses: Fu Yong Quah, Imperial College London, London, United Kingdom, fyq14@ic.ac.uk; David Thomas, Imperial College London, London, United Kingdom, dt10@ic.ac.uk.

XXXX-XXXX/2018/5-ART $15.00
https://doi.org/0000001.0000001

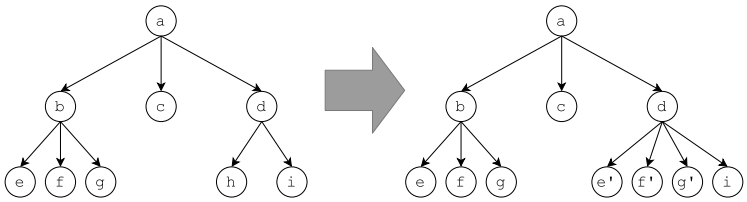


Fig. 1. The transformation as a result of $merge(b, h)$

as defined in the abstract. Stability (formally defined later) ensures that the label of a node is affected only by nodes that have been affected in earlier merge operations. The `gensym` function does not satisfy the stability property, as demonstrated by a simple example in figure 2. A naive implementation that copies the identifiers from the transitive closure of u directly to v does not mantain the uniqueness property across mutations, as shown in figure 3, as it is possible to inline the same function into two sibling declarations (For eg: a code snippet of the form $f(x) = g(x) + g(1 + x)$ where both instances of g are inlined.)

```
int g() { ... }
int f(int x) {
   return g(x + 1); // [1]
}
int main() {
  int a = g(2)   // [2]
        + f(2);  // [3]
  printf("%d\n", a); // [4]
}
```

```
int g() { ... }
int f(int x) {
   return g(x + 1); // [1]
}
int main() {
  int a = g(2)   // [5] (inline f(1))
        + f(2);  // [3]
  printf("%d\n", a); // [4]
}
```

```
int g() { ... }
int f(int x) {
  return g(x + 1); // [1]
}
int main() {
  int a = f(1)   // [3]
        + g(3);  // [5] (inline f(2))
  printf("%d\n", a); // [4]
}
```

Fig. 2. The numbers in square brackets in comments refers to the call site identifier. The centre and right-most column shows the modified code after different inlining decisions are taken from the code snippet on the left-most column. `gensym` cannot consistently label call sites across different set of inlining decisions.

A consequence of function inlining is that new variable names needs to be generated to avoid the name-capture problem. LLVM [4] ammends variable names by adding a suffix to SSA register names when performing function inlining. This problem, however, is generally of greater interest in high-level compiler passes on ASTs of tree-based IRs. The Flambda pass (with a tree-based IR) in the OCaml compiler [5] uses an approach similar to `gensym` to generate unique names. The Glasgow Haskell Compiler (GHC) [2] takes the most interesting approach towards variable renamining on function inlining in the GHC Core Pass (which is also a tree-based IR), by using trees [1] to generate new variable names [3]. All the above works have primarily focused on generating new locally unique variable names, without any concern on stability or the ability to reference them when subjected to different set of manipulations. Based on our knowledge of related work, a generalised stable labeling for AST nodes (inclusive of variable names and call sites) has not been formally studied.

2 NOTATION CONVENTIONS

Some relevant sets are defined as follows:

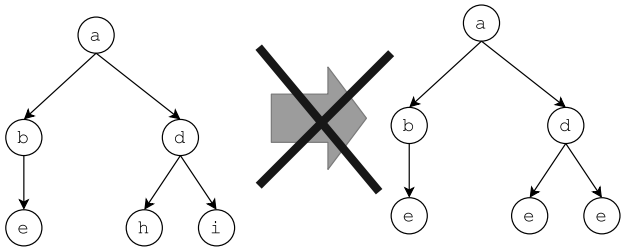


Fig. 3. A naive id assignment for merge operation that does not work. Replacing nodes e and nodes f with the contents of node b results in a conflict in labels.

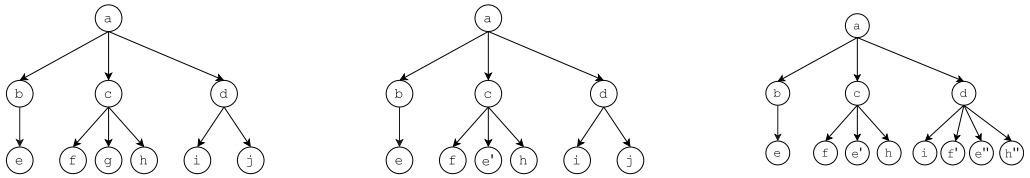


Fig. 4. Trees that will be used as running examples to explain algorithms

T	Set of all AST instances
V	Set of all vertices in all possible ASTs.
$C(v)$	Set of child vertices of a node v.
$M(d, a)$	Set of pairs (v, v'), where v is a child of d and v' is generated from merging.
$\mathcal{I}$	Set of all identifier objects.

The literal (possibly with subscripts) d will commonly be used as the source of a merge operation and a will be used to refer to the merge destination. v will be used as children of d whereas v' will be referred to as the newly created nodes after merging d into a, that is the set of nodes that replace a upon merge. Greek letters (ϕ, θ, γ) will be used to refer to elements of $\mathcal{I}$. For simplicity, $\mathbb{Z}$ will be used to label arbitrary stamps. In practice, the set of identifier is more likely to exhibit the form $(CompilationUnit \times \mathbb{Z})$. Taking inspiration from haskell syntax, the notation $[a]$ will be overloaded to refer both a singleton list a and the category of lists of type a; the : operator will be used to appending an element to the head of a list.

To ease illustration, the trees in figure 4 will be used as running examples, and referred to as t_0, t_1 and t_2 respectively. t_1 and t_2 are obtained from single merge operation on t_0 and t_1 respectively.

3 PROBLEM DEFINITION

3.1 AST Construction

ASTs that require labeling will be constructed in three phases.

3.1.1 Initialisation. Labels are not required for nodes at this stage. The AST can be arbitrarily constructed.

3.1.2 Fixation. All nodes are assigned a unique stamp using any methods, eg: using the the `gensym` function. After this phase, arbitrary tree manipulations are no longer legal.

3.1.3 Stable Manipulation. A tree instance can be modified with the following operations:

- Removal $remove(tree : T, u : V) \rightarrow T$ - Nodes can be removed without any restrictions.
- Merge - $merge(tree : T, u : V, v : V) \rightarrow T$ is defined as an operation that replaces node v with the transitive closure of node u, but excluding node u itself. For example, t_1 is obtained by running a merge operation on t_0, essentially replacing node g with the transitive closure of node b. The original node is not modified by this merge operation, other than the field denoting its list of children. The identifiers (or relevant intermediate data structures) are incrementally updated with merge operations - That is, node e does not necessarily have the same identifier as node e'.

The initialisation and fixation phase correspond to parsing a source file for the AST prior to any compilation passes. The stable manipulation phase corresponds to compilation and optimisation passes.

3.2 Functional Requirements

The algorithm is required to define an *id* function, which will be used upon fixation of the tree's initial tree. It has to fulfill the following logical properties.

PROPERTY 1. ***Mutation unrolling*** *where a is the merge destination of some source node d, $\exists(f,g).\forall(v,v') \in M(d,a).g(id(v')) = f(g(id(a)), g(id(v)))$. The identifier yields information about the total amount of merge that was required to arrive at a node.*

PROPERTY 2. ***Stability*** $\forall u \in V.\forall v \in C(u)$. *$id(v)$ is independent of $\{id(v') \mid v' \in C(u) \setminus \{v\}\}$ and its descendants. This ensures that the identifier of a node independent of unrelated merge operations.*

PROPERTY 3. ***Local uniqueness*** *After an arbitrary number of merge transformations, $\forall d \in V.\forall v \in C(d).\forall v' \in C(d) \setminus \{v\}.id(v) \neq id(v')$.*

It is trivial show that node removal will retain all of the above properties on a correctly labelled tree. Merging is the main transformation that the algorithm requires special care studying.

4 ALGORITHM

Consider AST nodes V and identifiers $\mathcal{I}$ with the following type definitions:

```
data V  = Apply Id | Decl Id [V]
data Id = (Int, [Id])

id (Apply x) = x
id (Decl x _) = x
children (Apply children) = []
children (Decl _ xs) = xs
update_id (Apply id) id' =
  Apply id'
update_id (Decl id xs) id' =
  Decl id' xs

stamp (x, _) = x
parents (_, p) = p
```

$V = Apply\ \mathcal{I} + Decl\ (\mathcal{I} \times [V])$

$\mathcal{I} = (\mathbb{Z} \times [\mathcal{I}])$

$id : V \rightarrow \mathcal{I}$

$children : V \rightarrow [V]$

$update\_id : V \times \mathcal{I} \rightarrow V$

$stamp : \mathcal{I} \rightarrow \mathbb{Z}$

$parents : \mathcal{I} \rightarrow [\mathcal{I}]$

The main idea of the algorithm is to have the $\mathcal{I}$ data structure form a bottom-up representation of the set of merge operations that are formed. The algorithm comprises of two sections: (1) constructing the tree-form of the identifiers and (2) outputting the identifier as a list in a way which can be printed. The algorithms to generate identifiers are a result of such merge operations and unroll identifiers from $\mathcal{I}$ data structures are shown below and a haskell implementation is shown in 5. Both algorithms are algorithmically and conceptually very simple - the main challenge is proving their conformance with the above properties, whilst maintaining realistic memory footprints.

ALGORITHM 1: Creates a set of nodes with rewritten node identifiers upon merge.

Input: Function declaration, d and call site a
Output: $[V]$ a list of nodes with rewritten identifiers.
Function `Merge`(d, a)**:**

$acc \leftarrow []$
foreach $child$ ***in*** $children(d)$ **do**
  $id_{child} \leftarrow id(child)$
  $parents' \leftarrow id(a) : parents(id_{child})$
  $id'_{child} \leftarrow (stamp(id_{child}),\ parents')$
  $acc \leftarrow concat(acc,\ [update\_id(child,\ id'_{child})])$
end
return acc

ALGORITHM 2: Recursively unrolls the identifier from the representation.

Input: Node label $\mathcal{I}$
Output: $[\mathbb{Z}]$ a list of stamps corresponding to the identifiers specified above.
Function `Unroll`(id)**:**

```
/* Semantically equivalent to concat(concat_map(Unroll, parents(id)),
   [stamp(id)])                                                    */
```

$acc \leftarrow [stamp(id)]$
foreach $parent$ ***in*** $reverse(parents(id))$ **do**
  $acc \leftarrow concat(Unroll(parent),\ acc)$
end
return acc

```
merge d a =
  map (\child ->
      let ps = parents (id child)
          ps' = id child : ps
          id' = (stamp (id child), ps')
      in update_id child id')
    (children d)
```

```
unroll id = unroll' [] id
where unroll' init id =
  foldr (\p acc -> unroll' acc id)
    (id stamp : init) (parents id)
```

Fig. 5. Haskell implementations of the algorithm

Proposition 4.1. *The algorithm satisfies property 1*

Proof. Consider $(v, v') \in M(u, a)$ for some (u, a). Now let $f = concat$ and $g = Unroll$

$$
\begin{aligned}
& g(id(v')) = f(concat\_map(g,\ parents(v')),\ [stamp(v')]) \\
\leftrightarrow\ & g(id(v')) = f(g(id(a)),\ concat\_map(g,\ parents(id(v))),\ [stamp(id(v'))]) \\
\leftrightarrow\ & g(id(v')) = f(g(id(a)),\ concat\_map(g,\ parents(id(v))),\ [stamp(id(v))]) \\
\leftrightarrow\ & g(id(v')) = f(g(id(a)),\ g(id(v)))
\end{aligned}
$$

In the base case, in a freshly initialised AST, $parents(\phi) = \emptyset \leftrightarrow Unroll(\phi) = [stamp(\phi)] = concat(concat\_map(Unroll, parents(\phi)),\ [stamp(\phi)])$.

□

Corollary 4.2. *Identifiers cannot be empty.*

Proposition 4.3. *The Unroll function will always terminate for all well-defined identifiers.*

Proof. $Unroll(x)$ will not terminate if and only if the the $parents(x)$ has a cyclic connection back to x. Hence, it suffice to prove that there is not cycles in the parents relation. This can be proven by induction. Assume there are no cycles in the directed graph defined by edges $\bigcup_{u \in V} \{\ (u, p) \mid p \in parents(u)\ \}$, that is the edges define a Directed Acyclic Graph (DAG). Identifiers generated as a consequence of merge operations adds new nodes to the existing identifier's graph, maintaining the graph's acyclic-ness. Since the base case is an identifier with no outgoing edges, hence a DAG, the proposition is inductively proven. □

Proposition 4.4. *The algorithm satisfies property 2.*

Proof. In the proof of proposition 4.3, it has been shown that the forms a DAG. On detailed concrete observation, it is trivial to notice that the DAG points strictly towards parents. Hence, the $\forall\phi.Unroll(\phi)$ is dependent only on the node itself and parent nodes and hence, independent of its sibling nodes and descendants. □

Proposition 4.5. *The algorithm satisfies property 3.*

Proof. We first define identifier equality as $\forall x \in \mathcal{I}.\forall y \in \mathcal{I}.x = y \leftrightarrow Unroll(x) = Unroll(y)$, with the later equality following conventional list equality semantics. This can be proven inductively. First, consider two identifiers $id(a_x)$ and $id(a_y)$ with common prefixes. This denotes that they originated from a common sequence of merge operation. The inductive assumption states that $prefix(id(a_x)) = prefix(id(a_y)) \implies suffix(id(a_x)) \neq suffix(id(a_y))$, where for some suffix and prefix functions such that $Unroll(id(a)) = concat(prefix(id($
The proof proceeds by attempting to contradict this statement. Consider the case where $|Unroll(id(v_x))| = |Unroll(id(v_y))|$. By merging some u into a_x, some $v_x^{(i)}$ is unveiled. Granted that $|Unroll(id(a'_y))| < |Unroll(id(a_y))| \implies id(a'_y) \neq id(a_x)$. Now by merging u into a_y, $v_y^{(i)}$ is obtained where $\forall i.|v_x^{(i)}| = |v_y^{(i)}|$ For $|Unroll(id(v_x^{(i)}))| = |Unroll(id(v_y^{(i)}))|$, it is necessary for $suffix(id(a_x)) = suffix(id(a_y))$. The requirement contradicts with the initial assumption, and hence, the assumed assumed argument is unsatisfiable. By contradiction, it is shown that if the uniqueness constraint hold $\forall a \in V$ where the length of identifiers for its children are equal, then it also holds in the general case. The base case is once again yield by the case where no functions are inlined, of which the uniqueness is guaranteed by the algorithm's assumption about unique id_{node}. □

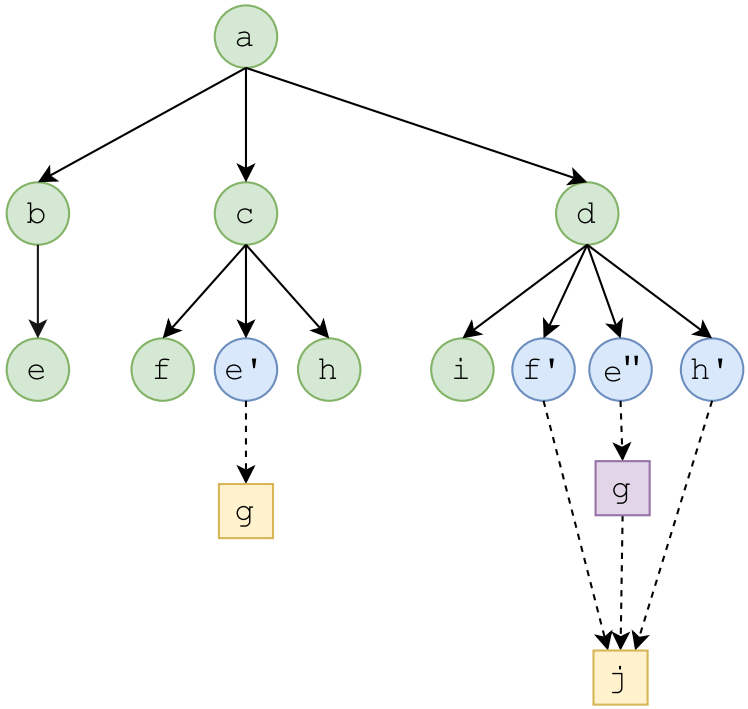


Fig. 6. Round nodes are nodes in t_2 and green nodes are nodes originally from t_0. Square nodes are removed from merge operations, but their identifiers are kept due to the recursive definition of identifier data structures. Note that the purple and orange g nodes occupy the same location in memory.

As all the propositions and corollary presented above were shown inductively, a very useful result follows: It is possible to run this merge for an arbitary number of times whilist retaining the properties in propositions 4.1, 4.3, 4.4 and 4.5.

5 DISCUSSION

5.1 Memory Footprint

It is trivial to show that creating a new identifier uses $O(1)$ memory, based on algorithm 1. Consequently, the memory requirement for storing the entire tree is $O(N_{merge}|C(v)|_{average})$.

What is less obvious is that the $\mathcal{I}$ recursively defines a bottom-up representation of merge operations conducted on the AST. Walking from a node up to the root of a mutation tree unveils the source of merge operations that has been undergone to unveil a certain node in the tree. The merge tree, as a result of merge operations carried out on t_2 is shown in figure 6. By traversing through the dotted edges, the original node that was removed through merge operations can be easily identified.

A naive equality implementation for equality check is to $eq(x, y) = eq(Unroll(x), Unroll(y))$. This uses linear-memory and would most likely suffice for general scenarios. However, when the size of the $\mathcal{I}$ tree is big (due to very aggressively inlining), a more memory efficient variant of an equality check can be performed by simultaneously walking both tree from the two given identifiers.

5.2 Globally unique identifiers

Property 3 defined above only guarantees the identifier uniqueness locally between siblings. To obtain global uniqueness, the global identifier node can be defined a pair of the path

to the node and the node's identifier, namely $\mathcal{I}_{global} = [\mathcal{I}] \times \mathcal{I}$. For example, the unique global identifier of node e'' in figure 6 given by $id_{global}(e'') = ([id(a),\ id(d)],\ id(e''))$, which is formed by a combination of the path from the tree root to node e'' and $id_{node}(e'')$.

6 EVALUATION

6.1 Functional and Correctness

The authors have used the above methodology for labeling function applicaiton nodes in an AST to override compiler inlining decisions. Specifically, we have modified the OCaml compiler to take customized inlining decisions in FLambda, an optimising compilation pass, using the presented methodology to label call sites for arbitrarily nested inlined function calls. It is likely that this methodology can be used in labeling other forms of inlining in various programming languages, such as template inlining in C++ or type-class inlining in Haskell [7].

6.2 Performance

To measure the asymptotic behaviour of the algorithm, recursively merging the leaf child of a node by itself in an AST with 100 nodes and a branching factor of 3[1]. This AST mutation is similar to those when inlining a recursive function repeatedly in the declaration, akin a loop unrolling operation. The memory usage is measured using valgrind [6] and the execution time is using the bash time utility. The relative execution time and memory usage in tables 1 and 2 is obtained by dividing by the case where there is only a single merge operation.

Table 1. Relative Memory Usage

Merge Ops	***Our Work (Raw Pointers)***	***Our Work (Shared Pointers)***	***Naive Algorithm***
1	1.000	1.000	1.000
10	1.011	1.019	1.016
100	1.111	1.203	1.715
1000	2.074	2.966	63.492
10000	12.759	22.532	6164.270
50000	57.100	103.724	153866.345

Table 2. Relative Execution Time

Merge Ops	***Our Work (Raw Pointers)***	***Our Work (Shared Pointers)***	***Naive Algorithm***
1	1	1	1
10	1	1	1
100	1	1	1
1000	1	1	2
10000	2	3	73
50000	6	12	1469

[1]Source code for this benchmark is available at https://github.com/fyquah95/stable-tree-labeling

REFERENCES

[1] Lennart Augustsson, Mikael Rittri, and Dan Synek. 1994. Functional Pearl: On generating unique names. *Journal of Functional Programming* 4, 1 (1994), 117 123. https://doi.org/10.1017/S0956796800000988

[2] SL Peyton Jones, Cordy Hall, Kevin Hammond, Will Partain, and Philip Wadler. 1993. The Glasgow Haskell compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, Vol. 93.

[3] Simon Peyton Jones and Simon Marlow. 2002. Secrets of the glasgow haskell compiler inliner. *Journal of Functional Programming* 12, 4-5 (2002), 393–434.

[4] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. San Jose, CA, USA, 75–88.

[5] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier R my, and J r me Vouillon. 2017. https://github.com/ocaml/ocaml

[6] Nicholas Nethercote and Julian Seward. 2003. Valgrind: A program supervision framework. *Electronic notes in theoretical computer science* 89, 2 (2003), 44–66.

[7] Simon Peyton Jones, Mark Jones, and Erik Meijer. 1997. Type classes: an exploration of the design space, In Haskell workshop. https://www.microsoft.com/en-us/research/publication/type-classes-an-exploration-of-the-design-space/