# Optimising Clang compiler with auto-tuners leveraging Structured Bayesian Optimisation

Szymon Makula
Hughes Hall

UNIVERSITY OF CAMBRIDGE

*A dissertation submitted to the University of Cambridge in partial fulfilment of the requirements for the degree of Master of Philosophy in Advanced Computer Science*

University of Cambridge
Computer Laboratory
William Gates Building
15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom

Email: Szymon.Makula@cl.cam.ac.uk

June 16, 2017

# Declaration

I Szymon Makula of Hughes Hall, being a candidate for the M.Phil in Advanced Computer Science, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.

Total word count: 14,119

**Signed**:

**Date**:

# Abstract

Compilers transform code from one language to another and became a well-researched field of computer science with a compact theory. Code optimisation is in cases of mainstream compiler implementations for C/C++ supported in the form of predefined levels - in the case of Clang they are -O* optimisation flags. Those optimisations in Clangs are not monolithic but composed of a list of passes. In this work, I explore how the compilation can be tuned for a program with a usage of Structured Bayesian Optimisation - a novel extension to the classical optimisation technique that allows for faster convergence if provided with prior knowledge in form of a probabilistic model.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Compilers transform code from one language to another - most commonly from highly abstract language to machine code. Since the origin of first simple compilers in the early 1950s (by Grace Hopper, Alick Glennie and others [3]), they became a well-researched field of computer science with a compact theory that today allow building a compiler for any language quickly. Currently, however, this transformation became a merely one of the functions that compiler engines do. Optimisation is another one and in the case of mainstream compiler implementations for C/C++, such as Clang, predefined "-O*" optimisation flags are the most widely known. Nevertheless, there are many underlying passes that actually optimise code and may or may not indeed improve the code for a given metric. Exploration of the impact of changing those on binary size, compilation time, the execution time of the program etc. will prove useful in different use cases - recurring deployment, development processes and computation time-saving. As Timmons [4] showed modifying the order and quantity of optimisation passes can produce better results than those achieved using the predefined flags. The previously unexplored idea of running inliner pass multiple times (at the beginning and end) provides a great improvement as it can give more opportunities for other passes to optimise the code. The main problem in gaining insight into the passes' behaviour is the enormous search space that is difficult to cover with

brute force iterating.

BOAT framework [2] allows developers to build efficient bespoke auto-tuners, which enables the auto-tuner to converge its iterative evaluation significantly faster. The core of BOAT framework is a novel extension of the Bayesian Optimisation called structured Bayesian Optimisation (SBO), where SBO leverages contextual information in the form of a probabilistic model of systems behaviour. BOAT provides Probabilistic C++ library for building such a probabilistic model by developers. Adding structural information to a probabilistic model of the objective function in Bayesian Optimisation outperforms standard Gaussian processes by orders of magnitude. BOAT could help to tackle the problem of optimisation of a compiler, which has massive complex parameter space and compiler developers have already good intuition on how different optimisation parameters behave as we designed them.

Employed in this tool I try to create an auto-tuner for a program compiled with the widely-used compiler - Clang. The aim is to show that a probabilistic model with structural information that represents the behaviour of the compilation process is possible and more robust.

The report is divided into four main chapters - Background, Optimiser Phase, Structured Bayesian Optimisation and Evaluation. Background focuses on introducing and discussing every major component that is used during the course of this project (limiting to only information included in this report). The chapter starts with an introduction to LLVM and an important for this project tool - Clang. I also explain the impact of LLVM IR and AST on the design and functionality that this compiler infrastructure provides. Next, I very briefly remind Monte Carlo sampling and Principal Component Analysis only to go in depth over Bayesian Optimisation and its underlying probabilistic model - Gaussian Processes. Finally, I introduce the idea of Structured Bayesian Optimisation - an extension to the optimisation framework.

The rest of the report is structured in the following way - Optimiser phase chapter focuses on pinpointing where the optimisation opportunity within

the compiler lies. Then, I introduce the implementation of Clang that is responsible for orchestrating the whole compilation process and in particular the optimisation phase. Using the ideas from Clang and LLVM source code I propose a formal definition of general optimiser model that I later use to define my experiments. The end of the chapter discusses experiments and their properties related to what has been the subject of this chapter.

The Structured Bayesian Optimisation discusses this extension in the context of the experiments - I present a sample semi-parametric model. Then, I describe experiments that will lead me towards my goal. I discuss the space of the problem and ways to simplify it enough for the auto-tuners so they actually can achieve some improvement

The Evaluation chapter has a twofold task in my dissertation - it evaluates experiments but more importantly tells a story on how all the pieces presented in the way in this report come together to gaining an insight into the compiler. I aim to find a way to use Structured Bayesian Optimisation to tune compilation for a program. The tuning process is not general and applicable to all but should be general enough that the similar approach will yield good results too.

# Chapter 2

# Background

This chapter discusses tools and frameworks that I rely on throughout this project and briefly puts the problem that this report discusses in perspective. I start with an introduction to the compilers three-stage design model followed by the more specific description of the tools that are the basis of evaluations throughout my project - LLVM compiler infrastructure and its front-end Clang. While discussing these subjects, I present design features of both that lay the landscape for my optimisation endeavours specifically the optimiser phase of the three-stage compilation model. Next, I proceed to discuss the frameworks that I use to investigate this landscape, including Monte Carlo sampling, Principal Component Analysis, Bayesian Optimisation and its extension Structured Bayesian optimisation.

## 2.1 Introduction to LLVM and Clang

This section discusses Clang compiler and its underlying LLVM compiler infrastructure. First, I present the classical compiler design model and retargetability property that is the basis of LLVM design. Next, I briefly describe the character of LLVM IR and passes that allow for retargetability. I then present the structure of Clang compiler and expose optimisation opportuni-

Figure 2.1: Simple classical three-stage compiler design

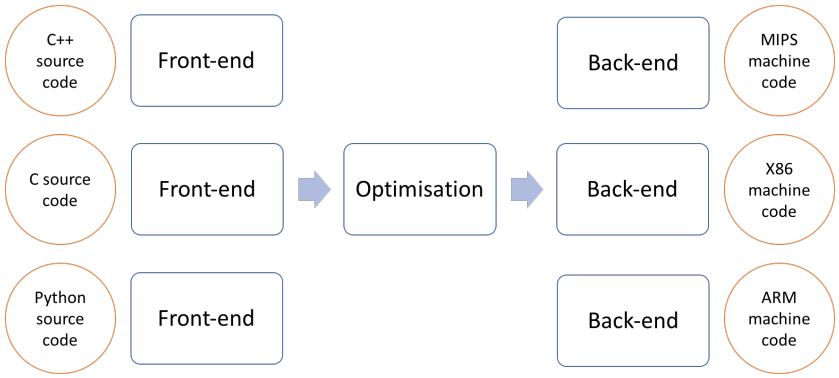

ties that it introduces in each phase. Finally, I briefly discuss the current choice of the heuristics used in the compiler phase that I focus on exploring throughout this project.

### 2.1.1 Classical Compiler Design

The design model most commonly used across static compilers composes of three major phases divided into the front end, optimizer and back end (sometimes referred to as code generation or code gen). The front end is responsible for parsing input source code to language specific Abstract Syntax Tree (AST) and checking syntax errors. In the second stage, optimizer takes the AST and applies optimising transformations to it. The final phase - back end - takes AST from optimiser and produces machine code exploiting specific target's features exposing optimisation opportunities. Note, the same model can be used for just-in-time (JIT) compilation and interpreters.

A modular implementation of this model does not reveal any additional benefits from a perspective of a single language and target compiler, yet today we have a wide landscape of languages that although are different would be compiled to the same target machine code. Furthermore, targets although support translation to different assemblies still will be translated from the same high-level languages. This dimensionality coupled with the modular design of the presented model reveals a retargetability opportunity of common optimiser that requires also a common and well defined Intermediate Representation. By leveraging retargetability we no longer need $N * M$ compilers

Figure 2.2: Retargetability of three-phase compiler design when each module is modular requires common intermediate representation.

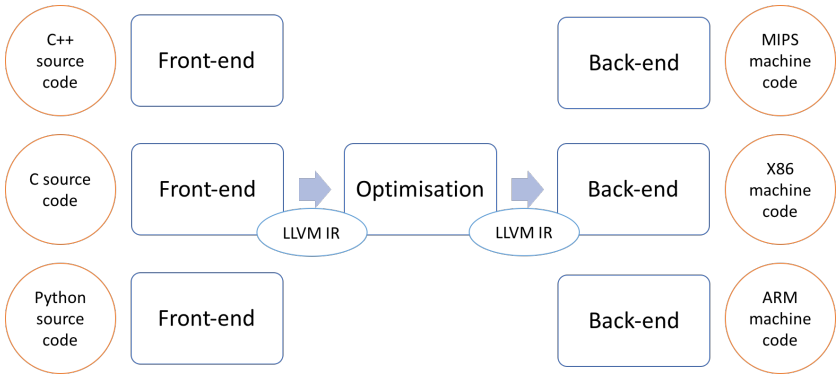



to support $N$ languages and $M$ targets. Instead, we only need one front end for each language and a single back end for each target. Since the optimiser stage is mutual across different languages and targets, the improvement of that stage will have a high impact on all compilers that rely on it. Another advantage of the model is a broader set of programmers who use it - for an open source project, a larger community of contributors will likely involve in the project which naturally leads to more enhancements and improvements to the compiler.

### 2.1.2 Introduction to LLVM

The LLVM project, started in 2000 at the University of Illinois at Urbana-Champaign, was originally an infrastructure developed for a research project to explore dynamic compilation techniques among static and dynamic typed languages. It distinguished itself with reusable libraries and well-defined interfaces [5]. Most of the compiler tools at that time were single-purpose monolithic programs very hard to reuse across different stages of the same compiler infrastructure and unsuitable for another purpose without major rework. Although, it should be noted that a few examples of successful modular model implementations existed then. But for different reasons were not suitable for full implementation of the three-stage pipeline [6]. The first example

Figure 2.3: LLVM uses its own abstract and machine-independent intermediate representation to achieve retargetability.




are Java and .NET virtual machines, both have a well-defined bytecode IR, provide just in time (JIT) compiler and runtime. However, the bytecode is specifically designed for JIT languages and the runtime support of garbage collection would cause the code to underperform if written in a language that does not match closely this machine model such as C (where memory management is manual). Another well known and popular implementation of the model is GCC which supports many front ends and back ends. Since GCC was originally designed and developed as a monolithic C compiler program, it is unrealistic to reuse parts of GCC compiler in a separate application without pulling in most of the project. Nonetheless, a number of efforts are constantly undertaken to modularise code. Currently, LLVM provides a well defined Intermediate Representation which I examine a bit closer in the next section and implements both - optimiser and back end compilation stages.

### 2.1.3 LLVM IR

At the core of portability, modularity and other benefits of LLVM lies the Intermediate Representation (IR) that it introduces - a low level, abstract, flat, machine-like programming language similar to assembly. LLVM IR abstracts away most target dependent features of a language, for example, it uses registers abstraction similarly to many assemblies. However, in contrast

to real word machine codes which have a limited number of registers, LLVM IR assumes unlimited count of single assignment registers. This assumption relieves the transformation on LLVM IR from unnecessary register juggling that would be target dependent - different number or type of registers may reveal additional optimisation opportunities that are not shared among all targets. Single assignment feature of a register makes data flow explicit - every consumer of an instruction result refers to the same output register. Note, that memory in LLVM, however, can be overridden.

```
int var = foo();
++var;
```

Listing 2.1: Sample C++ code

Sample source code from listing 2.1 in LLVM can be translated to LLVM IR shown in the listing 2.2

```
; int var
%var = alloca i32 , align 4
; var = 0
%0 = call i32 @foo()
store i32 %0 , i32* % a
; ++var
%1 = load i32 * % a
%2 = add i32 %1 , 1
store i32 %2 , i32* % a
```

Listing 2.2: Direct translation of each C++ instruction from listing 1

### 2.1.4 LLVM Abstract Syntax Tree and passes

Another important abstraction in LLVM is Abstract Syntax Tree (AST) - a tree representation of the abstract syntactic structure of source code. Although ASTs alone are part of many compilers, LLVM AST distinguishes itself with fully mapped AST to LLVM IR support (a sample code and its AST are shown in listing 2.3). This inverse relation allows for modification of IR while consuming AST and is supported in LLVM Pass Framework.

This framework plays the critical role in the LLVM system after LLVM IR, because it is the basic building block of most compiler parts which shape modular design of LLVM. Passes perform transformations and optimisations that build the compiler, they also conduct analysis of which results may be used by these transformations. For example, common expression elimination, as well as pessimistic routines that could be used to expose or analyse optimisation opportunities for follow-up passes - the combination of loop unrolling pass, followed by common constant propagation pass reveals data flow within the loop that allows for non-obvious otherwise common constant propagation. This procedure might produce bigger binary but decrease execution time depending on a user needs.

```
$ cat a.cpp
int foo();

int main() {
    int var = foo();
    ++var;
}


$ clang -Xclang -ast-dump -fsyntax-only a.cpp
TranslationUnitDecl 0x7ffd09017ed0 <<invalid sloc>> <invalid sloc>
|
... ; a few lines declaring some common global variables for the module
|
|-FunctionDecl 0x7ffd0a8000b8 <a.cpp:1:1, col:9> col:5 used foo 'int (void)'
`-FunctionDecl 0x7ffd0a8001b8 <line:3:1, line:6:1> line:3:5 main 'int (void)'
    `-CompoundStmt 0x7ffd0a800438 <col:12, line:6:1>
        |-DeclStmt 0x7ffd0a8003d8 <line:4:3, col:18>
        | `-VarDecl 0x7ffd0a800290 <col:3, col:17> col:7 used var 'int' cinit
        |   `-CallExpr 0x7ffd0a8003b0 <col:13, col:17> 'int'
        |     `-ImplicitCastExpr 0x7ffd0a800398 <col:13> 'int (*)(void)' <
    FunctionToPointerDecay>
        |       `-DeclRefExpr 0x7ffd0a800348 <col:13> 'int (void)' lvalue Function 0
    x7ffd0a8000b8 'foo' 'int (void)'
        `-UnaryOperator 0x7ffd0a800418 <line:5:3, col:5> 'int' lvalue prefix '++'
            `-DeclRefExpr 0x7ffd0a8003f0 <col:5> 'int' lvalue Var 0x7ffd0a800290 'var' '
    int'
```

Listing 2.3: First call to cat tool presents the content of the file. The next call to Clang, an LLVM frontend to C++ and compilation driver, prints the structure of AST for that function. Note that all nodes correspond to specific tokens in the source code. Furthermore, these nodes can be traced back to specific code locations in LLVM IR.

All LLVM passes are subclasses of the Pass class or, depending on their design, one of Pass subclasses - ImmutablePass, ModulePass, CallGraphSCCPass, FunctionPass, LoopPass, RegionPass or BasicBlockPass. Those

subclasses give the system more information about what a pass does, and how it can be combined with other passes. One of the main features of the LLVM Pass Framework is that it schedules passes to run efficiently based on the constraints that a pass meets (which are indicated by which class they are derived from). Subsequently, I briefly present those subclasses that I have encountered throughout the project. ImmutablePass does not change any state and never needs to be updated. It can provide information about the current compiler configuration or target but no analytical information. A pass derived from ModulePass uses the entire program as a unit, referring to function bodies in unpredictable order while adding and removing functions. Finally, FunctionPass is run on an individual function independently of any other code - it can only modify and understand only the function it is run on. Each of the passes may take additional variables that define and impact its behaviour.

A sample pass in listing 2.4 counts branch and switch instructions numbers across a module. A module is the top level container of all other LLVM Intermediate Representation (IR) objects. Each of the modules contains among others a list of global variables, a list of functions, a list of libraries it depends on and a symbol table. Both of those instructions that the pass counts are terminations (last instruction) of basic blocks - a container of instructions that execute sequentially (intuitively a node in control flow graph). The sample pass traverses every function in the module within which it visits every basic block for which in the if statements it checks the kind of that block's terminator instruction and updates counts accordingly. This pass does not make any modification and could be a subclass of ImmutablePass.

```
1 Module M;
2 unsigned int branch_instr_count = 0;
3 unsigned int switch_instr_count = 0;
4
5 for (Function &F : M) {
6     for (BasicBlock &BB : F) {
7         if (BranchInst *BI = dyn_cast<BranchInst>(BB.
    getTerminator())) {
8             if (handleBranchExpect(*BI))
```

```
9                   branch_instr_count++;
10          }
11          else if (SwitchInst *SI = dyn_cast<SwitchInst>(BB.
    getTerminator())) {
12              if (handleSwitchExpect(*SI))
13                  switch_instr_count++;
14          }
15      }
16  }
```

Listing 2.4: "Meat" of sample LLVM pass that counts branch and switch instructions number in a module.

### 2.1.5 Introduction to Clang

Clang is a popular front end atop LLVM framework for C language family including C++, OpenCL and others which parses the source code to LLVM IR. Clang is also a driver that orchestrates the whole compilation process starting from arguments collection and processing, through syntax error checking and source code to LLVM IR parsing to optimiser and back end. The latter is achieved through PassManager and PassManagerBuilder classes that can be used to control the optimising and code generation phases. Namely, each stage consists of a list of passes that transform original code to its final form. Optimiser pass always has Target Transform Information and Target Library Information passes that provide necessary information for linking. In case of code generation phase, a few passes generating and optimising the machine code are necessary per target. The project aims to be fully compatible with GCC in long term and its command-line interface is already very similar to and shares many flags and options with GCC.

Clang supports a few predefined optimisation flags compatible with GCC - O0, O1, O2, O3 (Ofast) and Oz (Os). O0 means that no optimisation is applied - the optimiser will only execute the passes necessary for linking and it is the fastest compilation that can occur. O1 is supposed to generate code faster execution-wise than O0 but slower than O2 which in turn is supposed

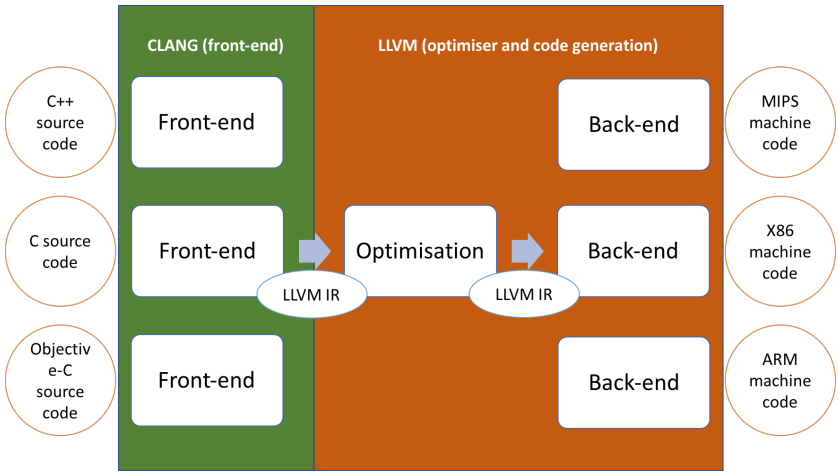
Table 2.1: Part 1: The list of the passes that transform the code and are part of at least one of the predefined optimisation flags

| Pass Name | Type | Args Count |
|---|---|---|
| Aggressive Dead Code Elimination | FP | 0 |
| Alignment from assumptions | FP | 0 |
| Combine redundant instructions | FP | 1 |
| Conditionally eliminate dead library calls | FP | 0 |
| Dead Argument Elimination | FP | 0 |
| Dead Global Elimination | MP | 0 |
| Dead Store Elimination | FP | 0 |
| Deduce function attributes | P | 0 |
| Deduce function attributes in RPO | P | 0 |
| Delete dead loops | P | 0 |
| Early CSE | FP | 1 |
| Eliminate Available Externally Globals | MP | 0 |
| Float to int | FP | 0 |
| Force set function attributes | P | 0 |
| Global Value Numbering | FP | 0 |
| Global Variable Optimizer | MP | 0 |
| Induction Variable Simplification | P | 0 |
| Infer set function attributes | P | 0 |
| Interprocedural Sparse Conditional Constant Propagation | MP | 0 |
| Jump Threading | FP | 1 |
| Loop Distribution | FP | 0 |
| Loop Invariant Code Motion | P | 0 |
| Loop Load Elimination | FP | 0 |
| Loop Sink | P | 0 |
| Loop Vectorization | P | 2 |
| MemCpy Optimization | FP | 0 |
| Merge Duplicate Global Constants | MP | 0 |
| MergedLoadStoreMotion | FP | 0 |
| PGOIndirectCallPromotion | MP | 1 |
| Promote Memory to Register | FP | 0 |
| Reassociate expressions | FP | 0 |
| Recognize loop idioms | P | 0 |
| Remove redundant instructions | FP | 0 |
| Remove unused exception handling info | P | 0 |
| Rotate Loops | P | 1 |
| Simplify the CFG | FP | 1 |
| SLP Vectorizer | FP | 0 |
| Sparse Conditional Constant Propagation | FP | 0 |
| Speculatively execute instructions if target has divergent branches | FP | 0 |

Table 2.2: Part 2: The list of the passes that transform the code and are part of at least one of the predefined optimisation flags

| Pass Name | Type | Args Count |
|---|---|---|
| SROA | FP | 0 |
| Strip Unused Function Prototypes | MP | 0 |
| Tail Call Elimination | FP | 0 |
| Unroll loops | FP | 1 |
| Unswitch loops | P | 0 |
| Value Propagation | P | 0 |
| A No-Op Barrier Pass | MP | 0 |
| Function Integration/Inlining | P | 1 |
| Inliner for always_inline functions | P | 1 |
| Bit-Tracking Dead Code Elimination | FP | 0 |
| Promote by reference arguments to scalars | P | 1 |

Figure 2.4: In three-phase compiler model Clang is a frontend for C-family of programming languages and LLVM implements optimiser and backend. Note, that Clang is also a driver that controls runtime of those LLVM stages - LLVM on its own is a library.



to be also a bit slower than O3. The pass lists between O2 and O3 do not actually differ very much between each other - O3 only adds "Promote 'by reference' arguments to scalars" pass in comparison to O2. O3 and Ofast are equal to each other in terms of the pass lists for the optimiser phase.

## 2.2 Simple Monte Carlo sampling

One way to overcome the limitations imposed by high-dimensional volumes is simple Monte Carlo sampling. I discuss this method very briefly here and refer to [7] for more details. Monte Carlo sampling is a formal regime to explore domain spaces of a function $f : X \to Y$ with randomness. More intuitively, it is the formal definition of the natural selection at random variable $x \in X$ many times in order to estimate distribution of the function $f$.

## 2.3 Principal Component Analysis

Principal component analysis (PCA) is a statistical procedure that uses an orthogonal transformation to convert a set of observations of possibly correlated variables into a set of values of linearly uncorrelated variables called principal components (or sometimes, principal modes of variation). The number of principal components is less than or equal to the smaller of the number of original variables or the number of observations. I assume that this method is widely known and refer to [8] for in-depth exploration of the topic.

## 2.4 Bayesian Optimisation

In this work I use bespoke auto-tuners (BOAT) - a user defined auto tuner that underneath uses Structured Bayesian Optimisation extension to Bayesian Optimisation. This section introduces the basic concepts and ideas behind

this framework. I start with the presentation of Gaussian Processes, a low-level method in this methodology but important for the course of my investigation and explanation of the results that in Evaluation chapter. In my case, I will focus on Inference with this model as I use only this aspect of that model. I then proceed to discuss Bayesian Optimisation for which often Gaussian Processes are used as the model. Then, in the next section, I present Structured Bayesian Optimisation extension and show how it is implemented in BOAT toolkit that I use for some experiments.

### 2.4.1 Inference with Gaussian Processes

Gaussian Process (GP) is a probabilistic model defining a distribution over functions. GP can be used to formulate a Bayesian framework for regression. They have been used in a number of scientific fields but only recently were appreciated in machine learning [9].

Gaussian process defines distribution over function $f : X \to Y$ where $X = \mathbb{R}^n$, $\vec{x} \in X$ is a n-dimensional vector of real numbers and $Y = \mathbb{R}$, $y \in Y$ is a real number. It is fully defined by a mean function $m : X \to \mathbb{R}$ and a covariance function $k : X \times X \to \mathbb{R}$.

We define the Gaussian Process as:

$$f(x) \sim GP(m, k)$$

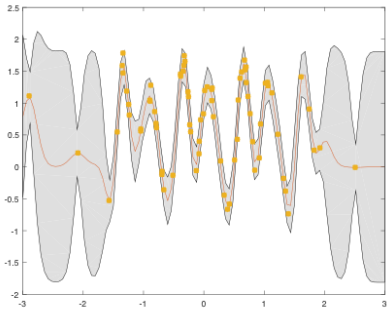
that can be read as "the function f is distributed as a GP with mean function m and covariance function k.

The covariance function describes how similar the two points are - using different ones will impact the properties of $f$ that the model will learn - we can use a simple squared exponential

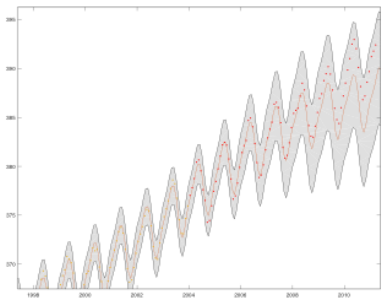
$$k(x_i, x_j) = \exp(-\frac{1}{2}|x_i - x_j|^2)$$

Figure 2.5: Two sample GP generated for given data points (yellow) in MATLAB using gp tool kit with different covariance function compose two contrasting models. Also, those model may coverage at various rate. The yellow curve represents mean and the grey segments, the region within which around 95% of GP distribution is located (twice the magnitude of standard deviation).

(a) Smooth covariance function



(b) Periodic covariance function



which will learn models that have continuous function and its local features well such as in 2.5. It might be a really good model if we do not have any additional prior knowledge - then this function could be just partially periodic and smooth otherwise. However, if we plug in this additional information to the model using a periodic kernel:

$$k_{periodic}(x_i, x_j) = \sigma^2 \exp(\frac{2\sin^2(\frac{\pi|x_i x_j|}{p})}{\lambda^2})$$

where $p$ is the period length and $\lambda$ is the lengthscale, we will get a model that expects periodic function and tries to fit it - 2.5. $\sigma$ and $\lambda$ are hyperparameters of this model.

Sampling from the prior distribution defined by Gaussian process for a set inputs $x_1, ...x_n$ requires us to first compute covariance matrix $K$:

$$K = \begin{pmatrix} k(x_1, x_1) & \cdots & k(x_1, x_n) \\ \vdots & \ddots & \vdots \\ k(x_n, x_1) & \cdots & k(x_n, x_n) \end{pmatrix}$$

and the mean vector $m$:

$$m = [m(x_1), ...m(x_n)].$$

With those terms defined, the values of $y_1, ...y_n$ corresponding to $x_1, ...x_n$ can be obtained by sampling from the multivariate normal distribution $y_i \sim N(m, K)$. The samples generated from the prior distribution are not useful on its own for the Bayesian optimisation, but they are the basic building block for defining the posterior distribution for another input $x_*$. Having a set of observed values for data points $x_1, ...x_n$ and $y_1, ...y_n$, we can define the posterior distribution for $x_*$:

$$\begin{pmatrix} y \\ y_* \end{pmatrix} \sim N(\begin{pmatrix} m \\ m(x_*) \end{pmatrix}, N(\begin{pmatrix} K & k_* \\ k_*^T & k(x_*, x_*) \end{pmatrix})$$

where

$$k_* = \begin{pmatrix} k(x_1, x_1) & \cdots & k(x_1, x_n) \end{pmatrix}$$

Then, the posterior distribution of $y_*$ can be analytically solved [9] with

$$p(y_*|x_*, x_1, ...x_n) = N(\mu(x_*), \sigma^2(x_*))$$

where

$$\mu(x_*) = m(x_*) + k^T K^1(< x_1...x_n > m)$$

$$\sigma^2(x_*) = k(x_*, x_*)k_*^T K^1 k_*.$$

As [2] highlights, an important intuition is that GP actually does not uses the true values of inputs. Instead, through application of covariance function on two inputs, GP knows how close they lie to each other relative to other pairs. I use this intuition when explaining some of the results in the evaluation chapter. Furthermore, the author notes that the mean vector can be moved off this definition

$$N(m, K) = m + N(0, K).$$

This property of GP shows that it does not learn the true values of the output - only how they differ from the mean.

### 2.4.2 Bayesian Optimisation

Finding minimum or maximum of a function $f$ is a common problem encountered in scientific fields and a wide range of tools exist to handle special cases of functions when the formula is known. For any differentiable function, we can compute its derivative, find roots of that derivative and pick extreme that we are interested in out of function values at those roots. However, the problem that we often face in engineering and sciences, be it computer or other fields, is when we do not have formula for the function that we want to optimise - let us call it a "black-box" function $f$.

Bayesian Optimisation was proposed as a solution to this problem in 1975 [10]. However, it became popular later when the method was combined with Gaussian Processes. The algorithm is particularly robust in the problems where sampling a value of the function is time-wise or computationally-wise expensive.

We want to find an extreme of an objective function $f : \mathbb{R}^n \to \mathbb{R}$ which for-

mula we do not know. Bayesian Optimisation method builds incrementally a probabilistic model sampling one value per iteration. The underlying probabilistic model is most commonly a Gaussian Process - although it should be noted that other models can be used and might perform better depending on the use case (random forests, neural networks etc.).

Bayesian Optimisation algorithm is in listing 1 [11]. The outer for loop can be run for a predefined number of iterations $N$ or some other condition could be used instead like model convergence etc. In the loop, we first carry out numerical optimisation of acquisition function $\alpha$ and sample the next candidate for a measurement $x_t$. This function represents belief that a sample over the objective function at point $x$ will improve the overall model $M$ - Intuitively, the higher value for an $x$ the more should value at $x$ improve the model $M$. In this step, for the numerical optimisation, we need to assume black box character of the acquisition function $\alpha$ and can use any method that applies in such case. Next, we make a measurement $y_t$ of the objective function $f$ at the sampled point $x_t$. Finally, we update the probabilistic model $M$ of the objective function $f$.

**Algorithm 1** Bayesian Optimisation algorithm

**Input:** Objective function $f : \mathbb{R}^n \to \mathbb{R}$
**Input:** Probabilistic mode $M$
**Input:** Acquisition function $\alpha : \mathbb{R}^n \to \mathbb{R}$
**Output:** Fitted probabilist model of the objective function $M$
  **for** $t \in (1, ...N)$ **do**
    $x_t \Leftarrow \arg\max_x \alpha(x|M)$ {Sample a point}
    $y_t \Leftarrow f(x_t)$ {Evaluate objective function at $x_t$}
    $M \Leftarrow M|(x_t, y_t)$ {Update the model $M$}
  **end for**

When choosing acquisition function, we want to trade-off exploitation, evaluating configurations which we know will perform well, and exploration while evaluating configurations which will be informative about the shape of the objective function. A number of functions can be used, I briefly present expected improvement function originally proposed by [10] as I used it in some experiments. For an input $x$s, it returns the expected value of the improve-

ment after evaluating $f(x)$ over the best objective function value found so far $\eta$ (incumbent):

$$\alpha_{EI}(x|\eta, M) = \int \max(0, m(x) - \eta) p(m(x)|M) \mathrm{d}m(x)$$

where $m$ is a single (sampled) model from $M$. 2.6 (borrowed from [1]) shows a visualisation of this algorithm's subsequent sample three iterations.

Some important limitations of Bayesian Optimisation include the problem of slow convergence or that the model does not capture enough of the objective function landscape when the input has too many dimensions - the curse of dimensionality consequence. Some of the methods to prevent these problems are discussed in [2]

## 2.5 Structured Bayesian Optimisation

Structured Bayesian Optimisation (SBO) is an extension of Bayesian Optimisation that improves its convergence using a user-given structured probabilistic model of the objective function instead of a generic model like a Gaussian process introduced in the work on BOAT in [2].

The methodology of an SBO is similar to the one of a traditional Bayesian optimisation and shown in the figure 2.7. Similarly, it performs three steps at each iteration:

1. It looks for a configuration with good predicted performance by the bespoke model
2. It evaluates the best found configuration using the objective function and collects runtime measurements
3. It performs inference on the model using all observations.

When compared with traditional Bayesian optimisation, using bespoke models brings three main advantages. First, it represents the users understanding

Figure 2.6: Figure borrowed from [1]. In the picture, the objective function is marked dotted curve - in real word application it is unknown. The figure shows the change of model, acquisition function and selection of the new candidate over three iterations of Bayesian Optimisation algorithm.

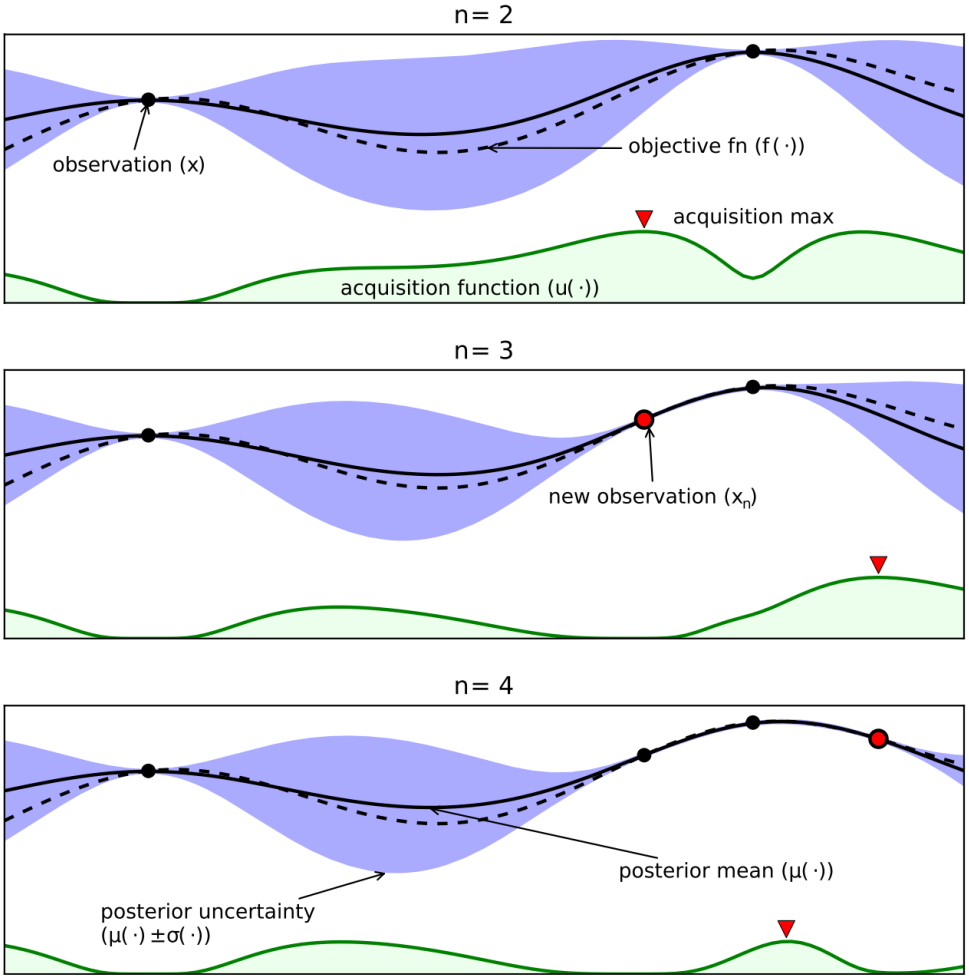
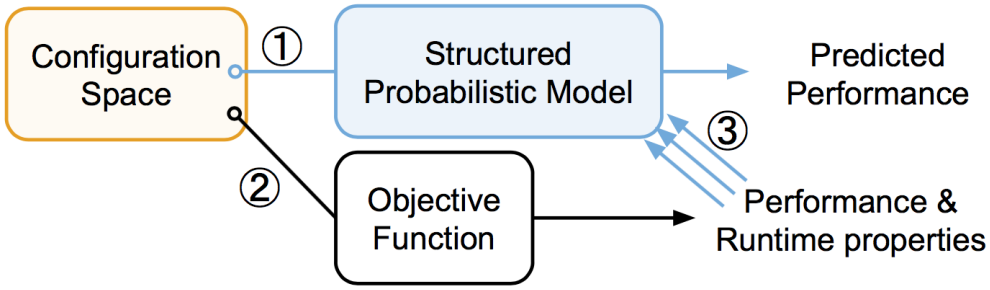


Figure 2.7: Figure borrowed from [2]. Three stages of the Structured Bayesian Optimisation along with the structured model that improves performance of the optimisation.




of the system behaviour introducing additional prior knowledge. Hence, it can drastically reduce the number of iterations needed for the model to converge towards the objective function. Second, it makes Bayesian optimisation applicable to new domains with complex configuration spaces, where a simple single model might not capture the behaviour of the system well. Third, using such a model allows us to collect many runtime properties reflected in the model and use them for inference.

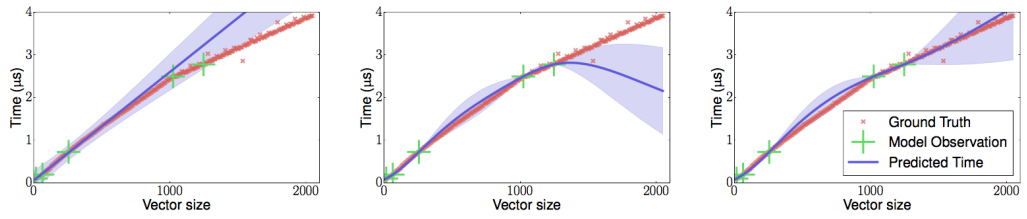
### 2.5.1 Probabilistic models

The author gives two recommendations for building such models. Firstly, the model should be compartmentalised - consist of a combination of independent components with each component predicting a single observable value. Secondly, each of those components should be a semi-parametric model. Those ideas are reflected in the implementation of BOAT and I discuss them in the following sections.

Let us consider a model of time it takes to insert an element into a sorted vector. The problem has complexity of $O(n)$ and we measure average time it takes to complete the operation.

By leveraging the knowledge we have about the model, its complexity, we can

Figure 2.8: Figure borrowed from [2]. Three models predicting the time to insert an element into a sorted vector after five observations.




(a) Parametric (Linear regression) (b) Non-parametric (Gaussian process) (c) Semi-parametric (Combination)

decide to use a parametric model of linear regression. In figure 2.8a we can see the result of fitting it after five iterations, the model tends to be under fitted - the general trend is captured, but important details in the behaviour such as smaller slope starting from around the size of 1000 is completely missed. Note, that the basic knowledge that we have does not help us in actually solving the problem well.

Instead, we can try to ignore the additional prior knowledge that we have and use Bayesian optimisation with a non-parametric model, for example, Gaussian process. The resulting mode after the same five iterations will look similar to 2.8b, the model is overfitted this time. However, after some number of iterations, it will converge to the correct shape except we did not take advantage of the additional knowledge that we have about the model.

Finally, we try applying a semi-parametric model combining parametric and non-parametric ones. Following the author "The non-parametric model is used to learn the difference between the parametric model and the observed data." The linear predict model is used to sample for acquisition function and the Gaussian process learns difference between it and the data. In BOAT the semi-parametric model is implemented in SemiParametricModel class.

BOAT also supports DAG models that can be used to create a combined model from semi-parametric models. I do not discuss them in here any further as I have not used them in my experiments for more details refer to [2].

```
1 struct GCRateModel : public SemiParametricModel<GCRateModel> {
```

```
2       GCRateModel() {
3           allocated_mbs_per_sec = std::uniform_real_distribution(0.0, 5000.0)(generator);
4           // Omitted: also sample the GP parameters
5       }
6
7       double parametric(double eden_size) {
8           // Model the rate as inversly proportional to Eden s size
9           return allocated_mbs_per_sec / eden_size;
10      }
11
12      double allocated_mbs_per_sec;
13  };
14
15  int main() {
16      // Example: observe two measurements and make a prediction
17      ProbEngine<GCRateModel> eng;
18      eng.observe(0.40, 1024); // Eden: 1024MB, GC rate: 0.40/sec
19      eng.observe(0.25, 2048); // Eden: 2048MB, GC rate: 0.25/sec
20      // Print average prediction for Eden: 1536MB
21      std::cout << eng.predict(1536) << std::endl;
22  }
```

Listing 2.5: Sample semi-parametric model implementaion

## 2.6 Previous work

In 2016 [4] the author conducts an investigation into the impact of reordering LLVM optimisation pass lists in Clang on compile time, build size and run time. The experiment consisted of an evolutionary algorithm - starting from a predefined list of passes it measures the times and size for that pass lists. Then, it iteratively repeats the following. First, it randomly chooses between adding a pass in random-chosen position, removing a random pass and swapping two passes in the list. Next, it measures compile time, build size and run time for this new pass list and if results are better than the best so far it accepts the new list and proceeds to the next iteration. Otherwise, with increasing probability, it rejects the worse results and changes the list back to this iteration original one and goes back. The author evaluates behaviour of three programs using this method - a popular video manipulation tool FFPMPEG, a lossless data compression program XZ and simple raytracer. The obtained results ranged from very promising with ray tracer where a significant improvement over the original pre-set optimisation O3 flag, through little variation in the behaviour for FFMPEG to no significant differences among results for XZ. Interestingly, the plotted values of measurements from

the experiment revealed some properties such as impact of early inliner on the ray tracer - for all features there was a better configuration than in those predefined.

Although the material for previous work I found is limited in the space of tuning - as there is no general method as of yet, I am aware that there were some attempts to tune compilers. However, none of them could have used Structure Bayesian Optimisation as it is a very recent work from Computer Laboratory at the University of Cambridge.

# Chapter 3

# Optimiser phase

Throughout this project, I focus on optimizer phase of the LLVM compiler infrastructure and Clang front-end for C++. In this chapter, I first briefly motivate this choice and discuss more specifically optimising compilation stage in the context of Clang. Next, I introduce the structure of a pass, present examples of each pass types and how pass lists are composed. Subsequently, I explore predefined optimisation pass lists - the heuristics that I treat as the baseline for any future evaluation. Finally, I formally propose the general optimisation phase model that I use throughout the project as the basis of models and experiments.

## 3.1 Introduction

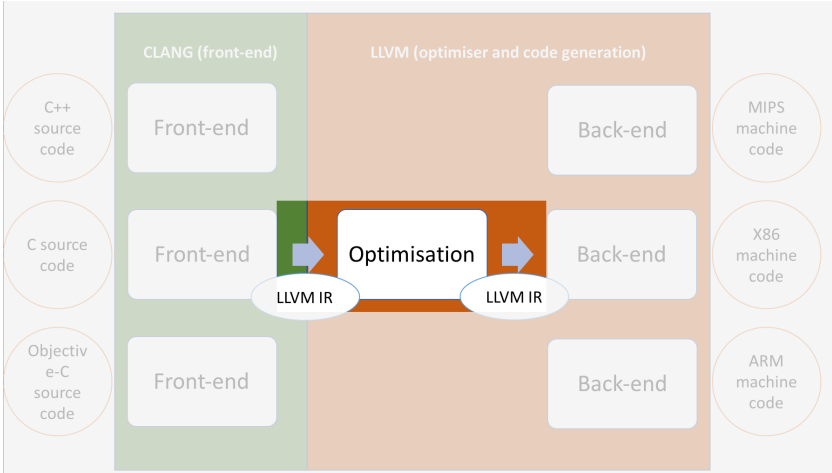
Clang and LLVM compiler infrastructure implement the classical three stage compilation model consisting of the front end, optimisation and back end phases (figure 2.3). This modular design is successfully implemented thanks to abstract, target-independent LLVM Intermediate Representation (IR) that serves as a bridge connecting the individual stages. Clang is an LLVM front end for C-like family of languages. Furthermore, Clang is a driver that orchestrates all compilation stages as LLVM on its own is just a library. However,

this library implements the two other stages of the compilation - optimisation and code generation. Both of those are made up of individual passes that transform LLVM IR code in some way. Since code generation is target specific, the only stage that is shared across the compilers that use LLVM is optimisation phase - that can be simplified to a list of passes that are subsequently applied to the output of a previous pass.

## 3.2 Motivation

Since the optimiser phase is centrally positioned in the compiler design model (figure 3.1) and is the only mutual part of the compiler used across different front and back ends, it seems natural that it should be of focus when trying to optimise a compiler. Indeed, the optimising stage of a compiler in case of LLVM is the most responsible for generating differentiable, in terms of performance, code - take for example a simple C++ program listed in the listing 3.1. It first initialises a vector of ten integers with the default value of 0 and prints them out. In this case, the vectors from C++ Standard Template Library are used in order to include more template code that preprocessor will insert while in the first stage of the program compilation. This code insertion should allow for more space for optimisation.The figure 3.2 shows that compiling the program at three different default optimisation levels O0, O3 and 0z produces significant differences between the generated program sizes. The improvement in size for this small and straight-forward program is in orders of 10%. Although the gains of this magnitude do not matter to individual users, for a big scale project and tools consuming significant amount of resources 10% can be a "big win." The improvement of order as small as 5% may have impact on profitability of certain projects, for example reducing an application size might significantly lower costs of the deployment if the number of users to deploy this application to is large enough. I discuss further improvements in real-world applications later in this chapter while describing modified experiment rerun from the previous work.

Figure 3.1: In this project I focus on optimiser phase as the exploration of this module can bring the most benefits to the overall performance of LLVM.



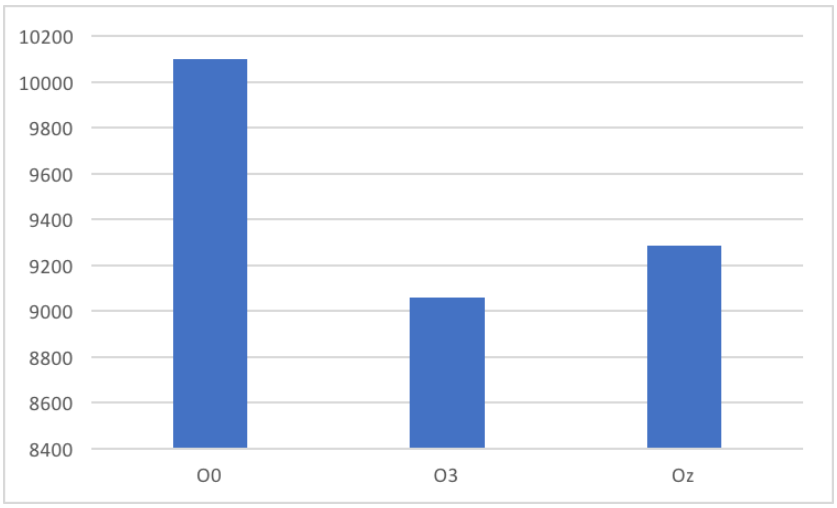

Optimising individual front-ends or back-ends will be applicable only to those languages and targets. However, due to specific features of languages or targets, these optimisation might be local whereas the optimiser phase focuses on leveraging LLVM IR features that are common across all compilers that use LLVM. Hence, I chose to focus on this particular stage when using LLVM with Clang frontend for C++. Furthermore, as shown later in the section discussing the implementation of Clang driver its optimising phase is the most dynamic one across the predefined optimisation phases.

```
#include <cstdio>
#include <vector>

int main() {
    std::vector<int> a(10);
    for (auto i : a) {
        printf("%d\n", i);
    }
}
```

Listing 3.1: Simple C++ program initialising a vector of ten with 0s and then printing the content of this vector.

Figure 3.2: The size in bytes of the produced binary for the sample code from listing 3.1 per chosen default optimisation levels - O0, O3 and Oz.




## 3.3 Clang driver

Except for being an LLVM front-end for C-family of languages, Clang is also the driver that controls the behaviour of the whole compilation process in the three-stage compiler model. The driver controls the execution of tools such as the compiler, assembler and linker. The subsequent phases that Clang runs are:

1. Preprocessing - it conducts tokenisation, #include expansion, macro expansion and other preprocessing (directives, etc.) on the input source file.

2. Parsing and Semantic Analysis - it parses the input file and translates preprocessor tokens into a parse tree. Subsequently, it applies semantic analysis on the parse tree to compute types for expressions as well and determine whether the code is well formed. If no errors were found, this stage outputs an Abstract Syntax Tree (AST).

3. Code Generation and Optimisation - it translates an AST into LLVM Intermediate Representation, applies optimising passes to that code and ultimately generates target-specific machine code from the ob-

tained LLVM IR. The output of this stage is an assembly file.

4. Assembler - it runs the target assembler to translate the output of the previous stage into a target object file. The output of this stage is an object file that can be used for final linking.

5. Linker - this final stage runs the target linker to merge multiple object files into an executable or dynamic library.

### 3.3.1 Clang Code Generation and Optimisation stage

In this project, I focus on the LLVM IR optimisation happening in the Code Generation and Optimisation step. The code controlling this stage can be found in EmitAssembly method of EmitAssemblyHelper (its signature is shown in listing 3.2). This function creates three PassManager objects that partially implement those stages of the compiler model and then runs them in the following order on the LLVM IR representation of the input source code - PerFunctionPasses, PerModulePasses, and CodeGenPasses. The PassManager class takes a list of passes, ensures they are set up correctly, and then schedules passes to run efficiently in the order they were provided.

```
1 void EmitAssemblyHelper::EmitAssembly(BackendAction Action, std
    ::unique_ptr<raw_pwrite_stream> OS);
```

Listing 3.2: Signuture of the lowest-level method responsible for controlling and running of Code Generation and Optimisation Clang stage.

PerFunctionPass object has only two modes in the default implementation of Clang. When flag O0 is enabled, the only passes added to this PassManager object are four analytical ones, subclasses of ImmutablePass, whose output is used in the subsequent stages. For any other optimisation flag (O1, O2, O3 and Os) apart from these four analytical passes another four optimising passes are added - Control Flow Graph simplification, Scalar Replacement of Aggregates, removal of redundant instructions (Early CSE) and a pass that lowers the 'expect' intrinsic to LLVM metadata (table 3.1).

Table 3.1: List of all passes in applied by PerFunctionPass PassManager for all but O0 predefined optimisation levels. the former four passes are analytical whereas the latter four conduct transformations.

| |
|---|
| Add Extension (early) |
| Library Info Wrapper Pass |
| Type-Based Alias Analysis |
| Scoped No Alias Alias Analysis |
| Simplify the CFG |
| SROA |
| Early CSE |
| Lower expect Intrinsics |

PerModulePasses is the pass list that is loaded with various configurations of 50 mutating passes (listed in tables 2.1 and 2.2) depending on the optimisation flags enabled. For example, when O3 is enabled, the list consist of 72 optimisation passes, subclasses of generic Pass class, FunctionPass and ModulePass, some of them repeating at different locations. In case of loop unrolling, it occurs twice in the O3 list - first time in the middle of the list - single value processing - and the second time at the end before the vectorization pass is applied that can take advantage of unrolling independent between iterations loops.

CodeGenPasses contains passes that optimise LLVM IR minding the target properties and leveraging its specific optimisation opportunities. The number and order of optimisation passes vary between different flags and is target-specific. The final two passes in this PassManager translate LLVM IR to the target assembly.

### 3.3.2 The optimisation pass list

Since PerFunctionPass has only two states - enabled or disabled optimisation - whereas CodeGenPasses and its optimisation are target-specific throughout this project I consider PerModulePasses as the only pass list to use for optimising. It also is the closest in resemblance to the optimisation stage from

the three-stage compiler model.

## 3.4 Optimiser general model definition

Having established understanding of Clang compilation and optimisation implementation, in this section I proceed to formally define the general optimiser model that will be the basis of future probabilistic models I present in the next chapter.

### 3.4.1 LLVM IR code

We define $S$ to be a set containing all possible finite LLVM IR source code that is syntactically correct. Note, that this set is infinite - if we take an element that is already in that set $s \in S$, we can generate through selection of a basic block in $s$ and adding to it a single instruction before a terminator in accordance LLVM IR syntax rules (in order to maintain syntactically correct code) to generate $s'$. Then, $s'$ is also an element of $S$ and for $s'$ - we can repeat the same procedure infinitely many times.

### 3.4.2 Pass

We have 50 passes that I consider in this project - full list in the appendix they were obtained by running all predefined optimisation levels and collecting all passes applied. We define $i \in 0, ...49$ to represent id of a pass. Then, we define configuration domain of the pass $i$, $C_i$ to be a set of all correct combinations of arguments represented as an integer vector of different length for each pass and $\vec{c_{i*}} \in C_i$ to be the default configuration of that $i$'th pass.

Now we define an $i$'th pass as a mapping $p_i : S \times C_i \to S$ that given some LLVM IR code and a configuration transforms it without changing it semantic behaviour. $P$ is a set of all passes defined - in this case $|P| = 50$.

### 3.4.3 Pass list

With the definition of the LLVM IR code and passes, we can finally define pass list as a finite mapping $l : \mathbb{Z}_+ \rightarrow P \times C$. Note, that the domain of this mapping has to be finite as otherwise, the compilation would never finish. Also, we define three helper functions that will be useful for defining algorithms - $passID : P- > 0, ...49$ that maps from a pass to its id, $pass : P \times C \rightarrow P$ that returns pass for a tuple of pass and configuration and $conf : P \times C \rightarrow C$ which returns the configuration for that tuple. This definition of the problem means that the space of all correct pass lists of maximum length $M$ size is lower bounded by $\sum_{k=1}^{M}(\binom{M}{k} * k!)$ for which lower bound would be $M!$. Hence, for an O3 predefined optimisation pass list, the lower bound of the problem space is of order 72! which is around $6.12 * 10^{103}$ - astronomically large.

### 3.4.4 Optimisation phase model

Finally, we can define optimisation phase in the form of the algorithm given in the algorithm listing 2. It requires two inputs - a non-empty pass list $l : \{1, ...N\} \rightarrow P \times C$ of length $N$ and LLVM IR source code to transform $s \in S$. With these arguments, the loop applies subsequent transformations (passes) in order they were presented in the pass list to the code along with the parameters that the pass takes. When the loop halts, $s'$ contains code after all transformations were applied. If the pass list $l$ is empty the output would be in $s$ - unchanged input source code.

## 3.5 Measuring performance of the pass list

The extreme predefined optimisation levels including, O0, O3 and Oz, are aimed at optimising, respectively, compilation time, run time and binary size. The objectives of those function naturally suggest what the optimisation

**Algorithm 2** Optimisation phase

**Input:** Pass list $l : \{1, ...N\} \rightarrow P \times C$
**Input:** LLVM IR source code to transform $s \in S$
**Output:** Transformed code $s' \in S$ after application of all transformation in the pass list $l$
  **for** $t \in \{1, ...N\})$ **do**
    $p_t, c_t \Leftarrow l(t)$ {Take a pass and its configuration}
    $s' \Leftarrow p_t(s, c_t)$ {Transform code with the pass}
    $s \Leftarrow s'$ {Update the previous LLVM IR code with new one}
  **end for**

objective should be. In this section, I briefly introduce how I measure each of those in the experiments that I conduct and what impact they may have.

### 3.5.1 Run time and compile time

Run time is a benchmarking feature that defines how long the execution of a compilation target takes on a representative sample for an input. Typically, the run time is the main concern whenever the additional time generates cost such as a server tool that decreased run time reduces needed server machines or applications where the real time aspect is crucial, for example, games or trading application. However, for some less intensive applications it might not be the most important aspect - take calendar for which even 50% decrease in run time would not matter (unless it is unusably slow in which case something is wrong with the application).

Compile time as a benchmarking feature might is most important to the developers who rebuild the source base they build often, for example when working on project which compilation time could be reduced even by only 10% and currently takes 15% of work time for an individual means already increase of 1.5% time of being able to work which might not be awful a lot for one person. However, for a team of 100 people, it translates in one day more of work.

In both of those cases, I use the same tool to measure times - Unix time

tool which executes and times a utility it is run with. I refer to user time as this value includes all threads and should be similar across different platforms with the same architecture and clock speed - almost independent of the number of cores and availability of threads.

### 3.5.2 Binary size

The final feature that I sample in the experiments is the size of the binary that a compilation produces. This is simply the size in bytes of that binary. As mentioned earlier the binary size matters in large scale deployments were small improvements in terms of size might give big wins for the whole infrastructure.

## 3.6 Pass ordering experiment

In this section I discuss the recreation of modified pass ordering experiment from 3, its purpose in my project is to gain insight into the behaviour of the programs, re-evaluate the findings from the mentioned work and obtain new pass lists exposing optimisation in respect to the predefined ones for further experiments.

## 3.7 Monte Carlo sampling of the configuration space per pass list

This section introduces the experiment that samples the parameter space with Monte Carlo methods and creates the baseline for experiments with models in the next chapter. In this experiment, we start with the pass list $l_0$ and default configurations for each of the passes in it. We also assume that $l_i$ does not differ in terms of domain and the passes it maps to between iterations $i$. However, the configurations to which it maps are varying. Algorithm

**Algorithm 3** The modified experiment from 3.

**Input:** Starting pass list $l$
**Input:** Program to compile $P$
**Input:** Probability $p$ that a worse result will be kept
**Input:** $p_\delta$ - the value by which the probability will decrease if worse result is kept
**Output:** Array containing results $r$

$l_0 \leftarrow l$
**for** $t \in \{1, ...N\})$ **do**
$l_t \leftarrow l_{t-1}$
Either remove a random pass from $l_t$ or add a random pass into random position of $l_t$.
$r_t \Leftarrow$ average results of measuring the compilation features after running optimiser algorithm on $A$ with the pass list $l_t$.
If measuring failed, repeat this iteration. {The list order is incorrect}
If build time is twice or more as long as the original one, repeat this iteration.
If results worse than the best and random test is within $p$, decrease $p$ by $p_\delta$ and proceed to the next iteration
**end for**

4 does $N$ iterations where at each we randomly sample configuration for the pass list and compile the program with that newly generated configuration of parameters. We save the results to the output array $r$.

**Algorithm 4** Monte Carlo sampling of parameter space

**Input:** Pass list $l$ to evaluate
**Input:** Program to compile $A$
**Output:** Array containing results $r$
  **for** $t \in \{1, ...N\})$ **do**
    **for** $i \in \{1, ...|l|\}$ **do**
      $c_{passID(i)} \Leftarrow$ uniformly at random select configuration from $C_{passID(i)}$
      $l_t(i) \Leftarrow (pass(l(i)), c_{passID(i)})$ {Update the mapping with new configuration}
    **end for**
    $r_t \Leftarrow$ average results of measuring the compilation features after running optimiser algorithm on $A$ with the pass list $l_t$
  **end for**

# Chapter 4

# Structured Bayesian Optimisation

This chapter describes three approaches and techniques I apply to model optimiser phase for Structured Bayesian Optimisation. I only focus on these three as they brought the most interesting results. The first two methods based on permutation ordering although turned out to be unsolvable with the frameworks I used, gave a compelling insight into the optimiser phase modelling problem nature. The third idea that I discuss is a built on top of the results from the first two failures and focuses on pass configurations rather than ordering.

In this chapter, I first motivate the idea of applying Structured Bayesian Optimisation to the Optimiser phase modelling. Then, I discuss the three approaches with details of how they represent and model that compiler phase in LLVM. Finally, the results and discussion about them is the main subject of the next chapter - evaluation.

## 4.1 Motivation

Bayesian Optimisation has been successfully used in parameter tuning of machine learning applications [12] and surpassed human experts. However, it struggles with high dimensional domains and might need a couple (too many) iterations to converge if the model tries to learn a complex function. Structured extension to Bayesian Optimisation proposed in [2] brings a chance for an auto-tuner to leverage the prior knowledge that a user has. This expertise can be modelled as the parametric function of the semi-parametric model and can be added as the process of creating auto-tuner iteratively progresses - a user can learn about some underlying correlations as they build and test the tuner. The second great benefit is the multilevel aspect of the model that can be achieved by creation of Direct Acyclic Graph models (DAGModel) which can model multilevel relationship between parameters and allow for larger number of the lowest-level parameters to be fed into the model. Optimiser phase of LLVM encapsulates pass lists chosen in heuristic way of which behaviour, we do not currently fully understand and for which no simple intuitive model exists, although refining this general optimisation problem in respect of reinforcement learning seems to be the most natural - refer to Future work.

## 4.2 Pass and pass groups order models

Work in [4] suggests that there is a potential for producing better code optimisation by changing the order adding and removing passes originating in a predefined pass lists. In this section, I discuss ideas and experiments to evaluate this idea with an auto-tuner.

### 4.2.1 Pass ordering model

We have a starting pass list $l$ - either a predefined or found through a process similar to the evolutionary algorithm from [4] and use it to define a set of all passes in that list:

$$W = \{\forall i \in \{1, ...|l|\} : (i, l(i))\}.$$

Now, we define a bijection $perm : 1, ...|W| \rightarrow S_{|W|}(W)$ that maps from an id to a permutation of the set of all passes. Note, that the number of permutations is $|W|!$ - same as the lower bound for the pass list space mentioned in the definition of pass list in chapter 3 and its magnitude of size is around $6.12 * 10^{103}$. This large search space cannot be searched with BOAT because C++, in which it is implemented, does not allow for such large numbers. Therefore, I choose 12 most significant passes from the list to be able so the id of the permutation can fit in C++ integer type.

To select those passes I use a heuristic based on the pass ordering experiment - I find the 12 steps that made the biggest improvement when added to the pass list in that experiment. I then proceed to build a simple generic Bayesian Optimisation model with BOAT to see if and how it converges, try to create semi-parametric model for SBO and Monte Carlo sampling of the space to have an idea of what the function looks like.

### 4.2.2 Pass groups ordering

Another approach that I explored is similar to the one presented above except I use all passes divided into a small number of groups. Then, I use the permutation number of that group as the parameter for Bayesian Optimisation. The hope here is to find groups of passes that applied will produce similar improvements. However, it is just a hypothesis.

## 4.3 Configuration models

The realisation of the problem with pass ordering, described in detail in the next chapter, calls for redefining the problem. Note, that the ordering tried to solve two problems at the same time - ordering and configuration of passes - the former one is very constrained in the above models. The set size of pass list, limited number of passes to be considered or lack of additional information about the properties of source code make from the search for ordering, a walk into the darkness with little feedback from the learning environment which comes naturally from the nature of Bayesian Optimisation.

For the sake of argument let us assume that we already are given the best pass list possible, then the problem becomes more natural to Bayesian Optimisation domain - we tune configuration values, which are following the definition from the previous chapter vectors of integers. The assumption we make can be thought to be fulfilled when we take the predefined optimisation pass lists (O3 and Oz) and some of the list passes found in the pass order experiment. Indeed, those pass lists are heuristically the best that out of the ones we are aware of.

The reduction of this problems leaves us with configuration vectors of integer values - yet, the total size of the vectors in case of 25 integer values for 72 passes in O3 configuration poses a problem with Bayesian optimisation which will struggle with effective exploration of such huge domain. Therefore, I run two experiments to reduce the problem.

### 4.3.1 Dimension reduction model

In the first method, I run Principal Component Analysis on the results from Monte Carlo sampling for configurations and reduce dimensions number by picking the top $n$ eigenvectors $\vec{v_1}, ...\vec{v_n}$ corresponding to $n$ eigenvalues with the biggest magnitude $e_1, ...e_n$. Then, I use these vectors $\vec{v_1}, ...\vec{v_n}$ as the basis of my new linear search space for Bayesian Optimisation and try to create a

semi-parametric model of it. An example of semi-parametric model used for optimising the run-time compilation feature follows the exponential function.

```
struct ParameterModel : public SemiParametricModel<ParameterModel> {
    ParameterModel() {
        alpha_ = uniform_real_distribution<>(0.0, 20.0)(generator);
        beta_ = uniform_real_distribution<>(5.0, 10.0)(generator);

        p_.default_noise(0.0);
        p_.mean(uniform_real_distribution<>(0.0, 9.0)(generator));
        p_.stdev(uniform_real_distribution<>(0.0, 1.0)(generator));
        p_.linear_scales({uniform_real_distribution<>(0.0, 160.0)(generator)});
        set_params(p_);
    }

    double parametric(double x1) const {
        return alpha_ * exp(-x1) + beta_;
    }

    double alpha_;
    double beta_;
    GPParams p_;
};
```

Listing 4.1: An example of semi-parametric model to reassemble exponential function

### 4.3.2 Benchmarking

In this section, I am using two benchmarking techniques and graphs to quickly show these features of the results. The first one is widely used when evaluating a tuner - the best result per an iteration. It shows how quickly a model converges and by taking average it can be used as a good benchmarking tool for that purpose. However, in my case, I am often comparing the behaviour of an auto tuner with exploratory techniques, such as pass ordering experiment, for which the comparison solely on the basis of the best value per an iteration is insufficient. Hence, I introduce cumulative probability distribution per feature value, which intuitively can be thought of as the probability that a test given a pass list and configuration selected at random from a sampling that the investigated process represents will produce a better result than the best produced with one of the predefined passes.

The second type of an evaluation technique that I introduced is a mapping of an individual compilation feature from a high-dimensional space to 2D or 3D using Principal Component Analysis. I use this evaluation technique to

quickly establish visually for a model if there is some parametric behaviour hidden within those dimensions. I conducted further analysis by pairing dimensions when no obvious mapping exists.

Finally, I introduce feature map, a 3D scatter graph that allows to quickly review the existence of the best configuration by visual inspection. Each compilation feature of a result - build time, binary size and run time are mapped onto axis, respectively x, y and z to represent one configuration point. Then, the closer a configuration representing point is the better overall performance.

# Chapter 5

# Evaluation

This chapter presents the series of experiments that ultimately led to achieving the project goal - showing that optimisation of a compilation with Structured Bayesian Optimisation is possible although as I point out it might not perform better in all cases without additional information. Note that I did not execute the experiments in the exact order as they were presented - many experiments did not produce the expected results. Therefore, I only discuss the "successful" ones. In this context, I treat an experiment to be successful if it can be used in the achieving the aim of this project, not necessary in a direct way - for example the Monte Carlo sampling experiment that originally was supposed to guide me into the right territory, turned out to also present a benchmarking value.

## 5.1 Aims

The goal of the project is to show that optimisation of a Clang compilation with Structured Bayesian Optimisation is possible. We already know that there is a space for improvement in regards to predefined passes from [4]. Therefore, I show in this chapter that:

1. Indeed, an evolutionary algorithm based on work from [4] can produce

better results than the predefined optimisation. However, it is an expensive and inefficient process. Furthermore, I show that this process cannot be optimised with Bayesian Optimisation in general.

2. I show that given a pass list (predefined or not), we can optimise the output code much more efficient and with better results than it was proposed. The result of few iterations can be used to establish a parameter space for optimisation

3. The Bayesian Optimisation can be applied to the compilation features optimisation and it converges faster.

4. The Structured Bayesian Optimisation can be applied and will converge faster than the standard one if the semi-parametric model is known. However, establishing the model requires either in-depth understanding of the individual pass impact that is difficult to model or sampling of the model using, for example, Monte Carlo method.

## 5.2 Design and Implementation

### 5.2.1 Clang optimiser

I implemented the formal optimisation phase model that I proposed in Optimiser phase chapter in Clang. The pass list is fed into the Clang as an argument containing the pass list file path - each line represents a pass, the first integer is the pass id, the second one is the number of arguments and they are followed by the values of these arguments if there are any. In terms of the PerFunctionPasses PassManager object, I enforce the eight passes to be run as mentioned in chapter 3. On the other side, PerModulePasses is fed with two required transformation passes and then with all passes from the file in the order as they appear. The rest of Clang behaves in the same way as it would be expected to allow for the usage of the same build systems as they would normally be used.

### 5.2.2 Experiment framework

For each experiment, I implemented a Python 3 framework which consists of a common file that implements the experiment and depends on Python libraries: sklearn [13] and numpy [14]. This file is shared across the configurations it is used to run on - programs, starting pass lists etc. for which a separate launch file is created. All shared resource like graph generation, common functions are located in modules within the common package that allows for an easy reuse. For the graph generation, I use a popular amongst data scientists plot.ly module. I run each of the individual measurements 10 times and take an average as the final output value for each compilation feature but binary size. Note, that for real word application this might not be realistic.

## 5.3 Common benchmarking properties

In all best feature per iteration graphs, the first value at point $x = 0$ is the original value for the starting point. Also, in these charts the lower the value in the earlier iteration, the better performance of the model.

For maps of pass configurations the closer a point to the origin - point $p = (0, 0, 0)$, the better is the configuration for that program.

When analysing cumulative probability distribution, the red vertical line in the chart marks the default value for this pass list. The higher probability at the point that intersects with the graph, the better the model. Another way of evaluating would be to take and compute expected value for that distribution.

In this evaluation, the default starting pass list is O3 unless specified otherwise.

### 5.3.1 Machine

All results that are presented in this dissertation are based on the results of the experiments that were run on macOS 10.12.5 with a 4-core 2.5 GHz Intel Core i7 and 16GB RAM. Some supporting experiments whose results I do not submit in this report were run on Ubuntu 16.04 equipped with a 4-core 2.8 GHz Intel Core i5 760 and 16GB RAM.

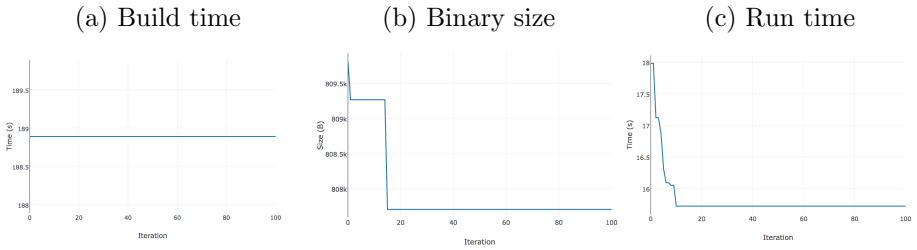
## 5.4 Test programs

During the project, I went through a number of programs including Google Proto Buffers, ffmpeg, wget, lepton and a few others less known applications like an open source ray tracer. In this evaluation section, I chose to focus on discussing experiments on Lepton and tar because they are very similar, both are compression tools, but yet very different, Lepton is single-purpose and tar is general-purpose. Furthermore, they were among the least computation expensive programs to rebuild and test which made experimentation easier.

### 5.4.1 Lepton

Lepton [15] is a C++ tool and file format for losslessly compressing JPEGs by an average of 22%. It achieves it by predicting coefficients in JPEG blocks and feeding those predictions as context into an arithmetic coder. Lepton is a great example of an application that is very specialised and would benefit from optimisation as it is also a tool created, open sourced and used by Dropbox Inc. in order to lower the size of the data storage for images their users store in their cloud service. Therefore, the tool is constantly used and probably deployed on many machines and faster runtime would be of value. I use a 12MB picture of the famous Cambridge view at King's College as the testing and conduct its compression with that tool. I measured standard deviation for the compression test $\sigma = 0.03$ seconds with the average runtime

Figure 5.1: The best result achieved so far per iteration starting from the O3 predefined configuration for lepton.




of $\mu = 12.51$ for one pass configuration means that error is less than 1% in testing. In case of compilation, the order of error is less than 0.2%.

### 5.4.2 Tar

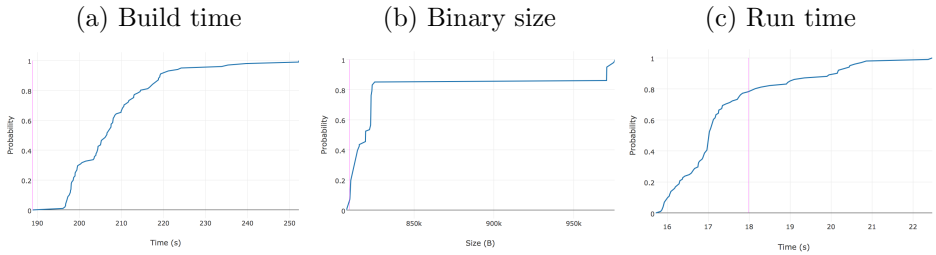
Tar [16] is a computer software utility for collecting many files into one archive file. It is also probably one of the most popular tools on Unix systems. Tar on its own does not compress, however, it uses additional libraries to do compression for it. I test the performance of tar by creating a new archive with zip compression out of a 290MB CSV file containing mix of string, integer and float data. Similarly to Lepton, the error rate is less than 1% with standard deviation $\sigma = 0.02$ and compilation timing error.

## 5.5 Pass ordering

The purpose of this experiment is twofold. First, it verifies what was achieved in [4] is indeed still possible, secondly, it provides a great benchmarking tool against other methods. The results in figure 5.1 show the best result achieved so far per iteration starting from the O3 predefined configuration.

In the figure 5.2, the original predefined pass list has the shortest build time. Although the charts suggest small wins in terms of the binary size and run time, they are not very significant. Note that improvements happen very

Figure 5.2: The distribution of individual compilation features in the experiment - the red line represents the original O3 performance for lepton.




rarely and the distribution chart confirms that - after almost 200 iterations the probability of better results is decent only for the run time feature and lies around 80%. Note, that those better results have around 10% speed-up compared to the default behaviour.

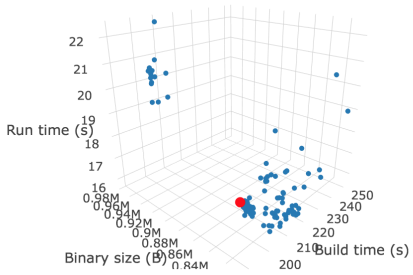
The poor improvements introduced by this method are probably caused by the huge search space that the algorithm tries to traverse - the domain size is lower bounded by 70!. The heuristic of this algorithm assumes that improvements lie close to each other in that space which is difficult to verify. The figure 5.3a seems to confirm that the problem lies in the too little space exploration - notice that points which represent a single pass list are clustered in two groups except for a couple ones lying slightly off.

## 5.6 Pass and pass groups permutation models

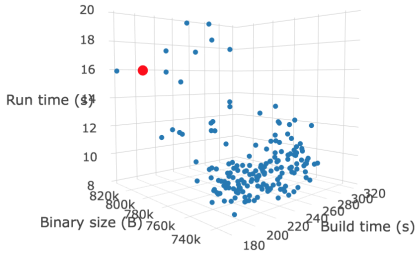
This experiment was run Boat and general Bayesian Optimisation was applied to learn if the model can be useful. The lack of any conclusive optimisation convergence pushed me to investigate deeper the model. I run a sampler of the subsequent permutation id feature measurements the results in terms of run time are in 5.4 on Lepton. In the chart a of this figure we can see that the behaviour of run time is highly volatile and expose neither any local nor global behaviour.

Figure 5.3: Distribution of compilation features per pass list - each point represents a pass list. (Lepton)

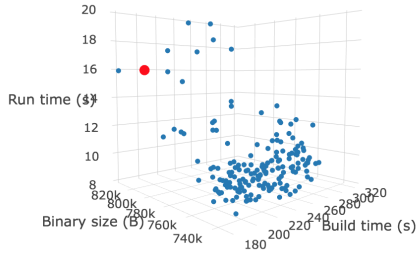
(a) Pass ordering




(b) Monte Carlo




(c) Binary size




(d) Binary size

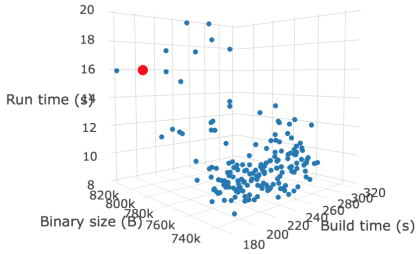
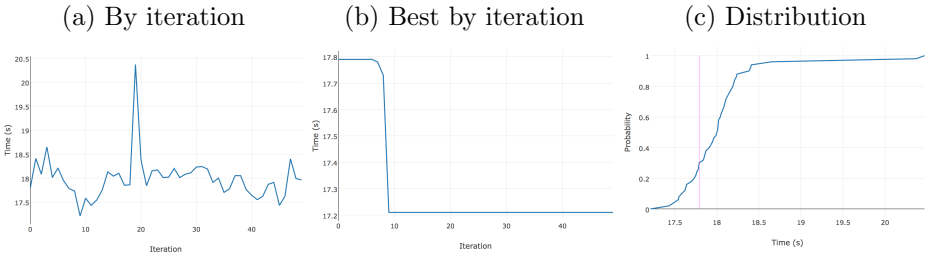


Figure 5.4: Run time feature for pass permutation experiment




This implicates that we need a better definition of permutation ID that would make the function smoother. However, since Gaussian Process, the probabilistic model in for BO, evaluates and learns a function's model though the perspective of covariance function, designing a better mapping from the permutation ID to a pass list that would make the function of a compilation feature smoother is equivalent to defining that covariance function. Yet, if we knew the covariance function $k$ that makes this compilation feature function smooth amongst the points that lie close each other in perspective of $k$ we have had solved the whole problem in the first place. Therefore, this approach is not going to work with Bayesian Optimisation and the model is discarded.

## 5.7 Configuration model

In order to reduce the parameter space, I first run the Monte Carlo sampling on the of all configurations for a predefined pass list. I then apply Principal Component Analysis to extract the most significant eigenvectors. In figure 5.5, the binary size and run time features seem to have a structure respectively, quadratic and exponential function. However, the build time feature does not appear to have any particular structure - potentially it has some in higher dimensional space. But I will ignore it and instead, I focus on exposing and evaluating the structure and performance with Bayesian Optimisation and its Structured extension.

Figure 5.5: Lepton compilation features in the reduced parametric space from the original 26-D space generated with Monte Carlo sampling of configurations.

(a) Build time (b) Binary size (c) Run time

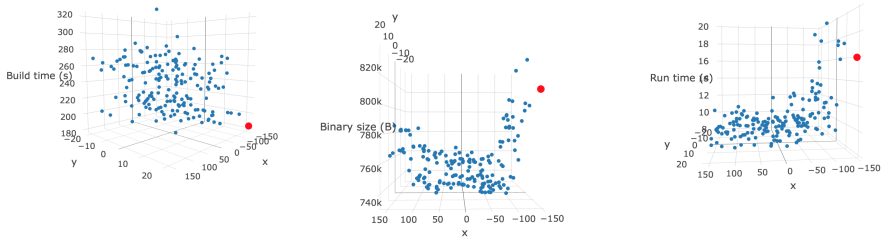
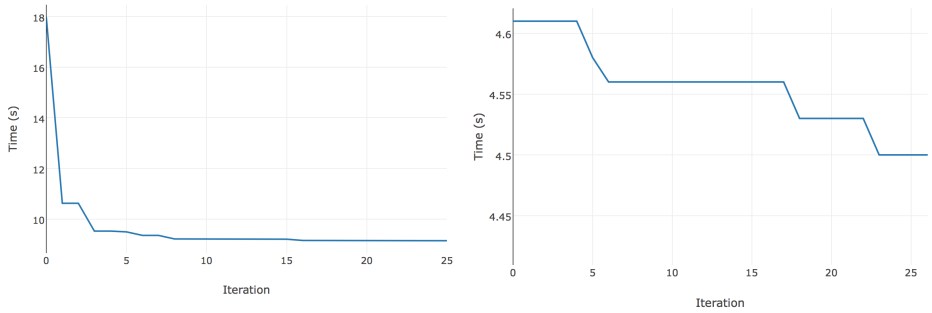



Figure 5.6: The best run time per iteration when using Monte Carlo on average.

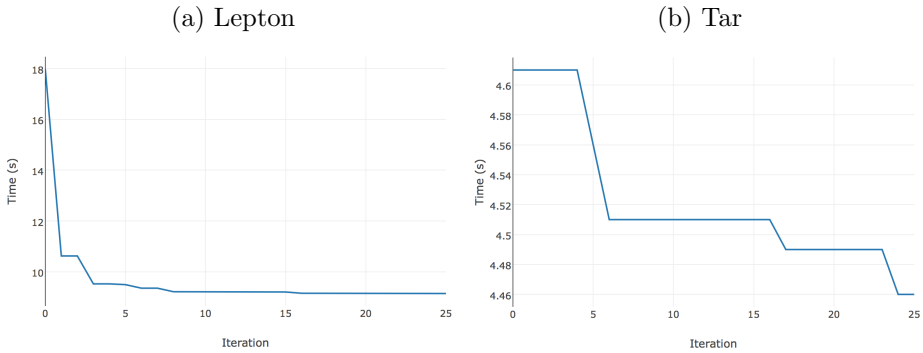
(a) Lepton (b) Tar




As the baseline for the next steps I set up to run both with Monte Carlo sampling of the parameter space and present the run time compilation feature results in figure 5.6. The purpose of running both programs with the same model is to establish whether one model per pass list is enough.

Both of the function seem to have roughly one-dimensional input. Therefore, I attempt Bayesian optimisation on the same parameter space and with the same pass lists for two programs Lepton and Tar. The figure 5.7 shows the convergence of each model. Lepton converges very fast but not faster than the Monte Carlo sampling and Tar is converging fairly well - definitely better than Monte Carlo sampling. Next, step is to try Structured Bayesian

Figure 5.7: The best run time per iteration when using general Bayesian Optimisation.



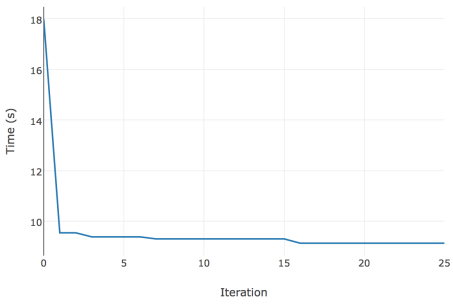

Optimisation.

I added semi-parametric model of an exponential function $f(x) = \alpha * \exp(x) + \beta$. Using the SBO extension causes both Lepton and Tar to flourish and achieve the best convergence so far. In case of the Lepton that is the expected result, yet for Tar which does not seem to have structure for the run time compilation feature - look at the figure 5.9. Therefore, I suspect that because the model distribution is flat and I was "lucky" the SBO worked well for Tar too. However, in less fortunate situation a separate model would be required for more efficient optimisation even though we optimise the same pass lists. Furthermore, this result suggests that there must be additional latent variables at play when optimising the compiler - I think that the features describing Abstract Syntax Tree and/or Control Flow Graph would have an impact in general optimisation. Note that the parameter optimisation method presented produced improvement of order of 50% when compared to the default. For a tool like Lepton that would mean half the number of servers needed to maintain the application running and half the cost. Also, this model found the best solution in comparison to the other models reviewed in this chapter.

Nonetheless, Tar has good structures for build time and binary size compilation features - figure 5.10.

Figure 5.8: The best run time per iteration when using general Structured Bayesian Optimisation with exponential function as the semi-parametric model.

(a) Lepton




(b) Tar

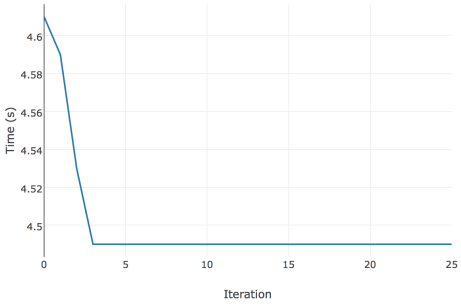



Figure 5.9: The tar does not have any structure in run times.

(a) Build time

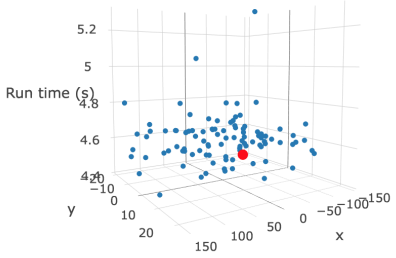
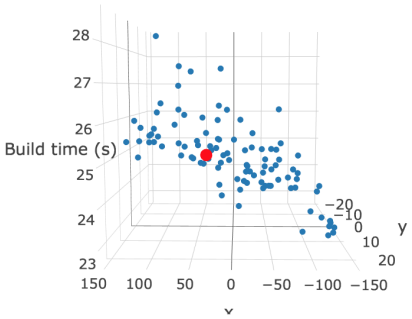
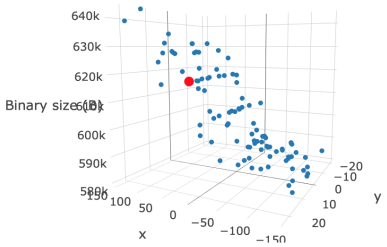



Figure 5.10: For other features tar has a good structure suitable for parameter optimisation with SBO.

(a) Build time




(b) Binary size



# Chapter 6

# Outlook and Conclusions

This chapter summarises and concludes the work done and described in this report.

## 6.1 Summary and Conclusions

In this report, I presented Clang, LLVM and optimisation opportunity lying in the optimisation phase of the three-stage compiler design. Furthermore, I proposed a formal general optimisation phase definition - that can be used for further design of models for Bayesian Optimisation and others. I showed how through Monte Carlo methods, Principal Component Analysis and Bayesian Optimisation along with Structured Bayesian Optimisation one can tune a compilation for specific program and bring even 50% gains in some cases. Also, I showed that the permutation model cannot be used together with Bayesian Optimisation. Finally, the results from analysis confirmed that using Structured Bayesian Optimisation brings additional robustness and converges in fewer iterations.

## 6.2 Future work

A few interesting directions in which this work could go are:

### 6.2.1 Control Flow Graph features

The tuning I presented in my work is specific per pass list and program that is tuned. Therefore, I suspect that there is some hidden structure in the code that could potentially allow for generic per pass list auto-tuning. If that is achieved, then a compiler would be auto-tuning itself and may find on its own a 50% speed gain opportunity. I conjecture that properties of Control Flow Graph [17] might help uncover this structure as passes often have impact on CFG - inliner explodes the code.

### 6.2.2 Reinforcement learning approach

The problem of finding optimal pass list could be redefined as a reinforcement learning problem [18] - given source code decide which pass to apply and with what value or no pass at all. However, this would be a really hard and time-consuming problem to solve.

# Bibliography


[1] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams, and Nando de Freitas. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, 2016.

[2] Valentin Dalibard. A framework to build bespoke auto-tuners with structured bayesian optimisation.

[3] John Backus. The history of fortran i, ii, and iii. In *History of programming languages I*, pages 25–74. ACM, 1978.

[4] Nicholas Timmons. Application performance and compilation time improvement through optimisation pass ordering in clang. To be published, currently available through Dr David Chisnall. Accessed: 2016-11-17.

[5] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM SIGOPS operating systems review*, volume 37, pages 29–43. ACM, 2003.

[6] Amy Brown and Greg Wilson. *The Architecture Of Open Source Applications*. lulu.com, June 2011.

[7] Christian P Robert. *Monte carlo methods*. Wiley Online Library, 2004.

[8] Ian Jolliffe. *Principal component analysis*. Wiley Online Library, 2002.

[9] Carl Edward Rasmussen. Gaussian processes for machine learning. 2006.

[10] J Močkus. On bayesian methods for seeking the extremum. In *Optimization Techniques IFIP Technical Conference*, pages 400–404. Springer, 1975.

[11] Eric Brochu, Vlad M Cora, and Nando De Freitas. A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599*, 2010.

[12] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.

[13] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine learning in python. *J. Mach. Learn. Res.*, 12:2825–2830, November 2011.

[14] Stefan van der Walt, S. Chris Colbert, and Gael Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science and Engg.*, 13(2):22–30, March 2011.

[15] Dropbox Inc. Lepton. https://github.com/dropbox/lepton, 2016.

[16] GNU. Gnu tar. https://www.gnu.org/software/tar/, 2017.

[17] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[18] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.