Enhancing Bayesian Optimization for Compiler Auto-tuning

Jiayu Zhao

University of Leeds

School of Computer Science

Submitted in accordance with the requirements for the degree of

Doctor of Philosophy

August, 2025

Intellectual Property Statement

I confirm that the work submitted is my own, except where work which has formed part of jointly authored publications has been included. My contribution and the other authors to this work has been explicitly indicated below. I confirm that appropriate credit has been given within the thesis where reference has been made to the work of others.

Chapter 4 is based on work published in:

- Jiayu Zhao, Renyu Yang, Shenghao Qiu, and Zheng Wang, "Unleashing the Potential of Acquisition Functions in High-Dimensional Bayesian Optimization", *Transactions on Machine Learning Research (TMLR)*, 2024.

Chapter 5 is based on work published in:

- Jiayu Zhao, Chunwei Xia, and Zheng Wang, "Leveraging Compilation Statistics for Compiler Phase Ordering," *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2025.

I confirm that I am the first author of all these jointly authored publications, that the work contained within these publications is directly attributable to me, and that the contributions of the co-authors have been in terms of general advice and assistance.

Acknowledgements

First of all, I want to express my deepest gratitude to my academic advisor, Professor Zheng Wang. His generous sharing of time, thoughtful insights, and unwavering support have been instrumental throughout my research journey. His patience, innovation, enthusiasm, and optimism have continually motivated and inspired me.

I am also truly grateful to every member of the DSS group at the University of Leeds. Your warmth and collaborative spirit have made this group a truly rewarding community.

I am indebted to all my friends in both the UK and China, whose encouragement and support have enriched my experience in ways too numerous to list.

I must also acknowledge the unconditional love and support of my family – my Mum, Dad, and sister Jin – whose faith in me has been a constant source of strength.

Finally, I would like to thank my girlfriend, Chenfei. Her endless understanding, patience, and encouragement have been my rock throughout this process. No words can adequately convey my gratitude for her support.

Abstract

Modern compilers offer a wide range of passes for code optimisation. Selecting the right combination and order of these passes, known as *phase ordering*, can improve the performance of compiled binaries. Autotuning, which refers to automatically searching the space of possible pass combinations, is a powerful technique for compiler phase ordering. However, its practical adoption remains challenging due to the vast search space of compiler optimisations and the high cost of evaluating candidate configurations.

This thesis enhances compiler autotuning by leveraging Bayesian optimisation (BO) to efficiently explore the complex space of compiler phase ordering. BO builds an online surrogate model to approximate the objective function to reduce evaluation overhead. It uses an acquisition function (AF) to guide sampling, improving search efficiency. While promising, applying BO to compiler autotuning requires addressing multiple open challenges.

First, standard BO struggles with high-dimensional search spaces like compiler phase ordering. To address this, this thesis introduces a simple yet effective AF initialisation strategy to enhance BO's ability to navigate high-dimensional optimisation spaces.

Second, the complex interactions between compiler passes make it difficult to model the relationship between pass sequences and performance to build an effective surrogate model. To tackle this, a new compiler autotuning strategy is proposed to incorporate compilation statistics to model these interactions. This method improves BO's search efficiency, requiring only one-third of the search budget compared to previous approaches while delivering higher-performance binaries.

Finally, a real-world program often contains multiple source files and complex compilation workflows. Applying compiler autotuning to such settings requires efficiently allocating the search budget across the compilation targets. To address this, this thesis presents an adaptive BO scheme that dynamically allocates search budgets across source files and develops a framework to automate compiler autotuning setup.

Together, these contributions improve the efficiency, scalability, and usability of BO-based compiler autotuning, making it a more practical tool for autotuning compiler phase ordering.

Contents

List of Figures

List of Tables

Abbreviations

BO	Bayesian Optimisation
GP	Gaussian Process
AF	Acquisition Function
EI	Expected Improvement
UCB	Upper Confidence Bound
GA	Genetic Algorithm
ES	Evolution Strategy
CMA-ES	Covariance Matrix Adaptation Evolution Strategy
DES	Discrete $1+\lambda$ Evolution Strategy
IR	Intermediate Representation
AST	Abstract Syntax Tree
CFG	Control Flow Graph
GNN	graph neural networks
RNN	Recurrent Neural Network

Chapter 1

Introduction

Compilers are the cornerstone of modern computer systems, serving as the essential bridge that translates high-level programming languages such as C/C++ into low-level machine instructions for execution. With the increasing complexity of hardware architectures and the growing diversity of software demands, compilers have evolved from basic translation tools to sophisticated optimisation frameworks, striving to maximise hardware performance. Over the decades, compiler design and optimisation have become critical for improving computational efficiency, reducing energy consumption, and enhancing overall system performance.

Modern compilers like LLVM [1] and GCC [2] provide a large number of optimisation transformations to select among. These transformations, referred to as *passes* in the compiler context, play a crucial role in optimising program performance. Each compiler pass performs a specific code analysis and transformation, including techniques like loop unrolling, instruction scheduling, and register allocation. The optimisation passes we enable, the order in which we apply them to a program, and the specific parameters of each applied optimisation pass have a substantial impact on program performance, such as execution time, code size, or power consumption. Compared to optimizing runtime or energy consumption, minimizing code size is generally a simpler task, as it primarily involves static properties of the compiled program. Code size can be evaluated without running the program, and is less sensitive to dynamic execution behavior or hardware-specific factors. In contrast, optimizing execution time or energy consumption requires modeling complex interactions between the program, compiler transformations, and the underlying hardware. These metrics depend on runtime behavior such as memory access patterns, branching, and instruction-level parallelism.

Especially, execution time is typically the top priority in industry, as faster programs directly improve user experience and system efficiency. Moreover, shorter execution time often leads to lower energy consumption, making runtime optimization beneficial for both performance and energy efficiency.

By default, compilers provide optimisation level settings, such as -O3 for execution time optimisation and -Oz for code size reduction, which apply a fixed pass sequence to enhance program performance on average. However, these default optimisation levels are not always sufficient for all programs. Each program has unique characteristics and may benefit from a different set of optimisation passes. For example, some programs may benefit from aggressive inlining that can increase register pressure but reduce function calling overhead, while others may benefit from register promotion to reduce memory access overhead or encourage vectorisation. Carefully selecting and ordering compiler passes to improve the performance of the generated code for a given program, known as the compiler *phase ordering* problem, has been an open problem in the field for several decades, and it is known to be NP-complete [3]. Compiler phase ordering is especially valuable for frequently executed programs. Even a minor enhancement in execution time can yield significant benefits over time.

Autotuning is an approach to address the problem of automatically selecting a good compiler optimisation configuration for an application. Typically, compiler autotuning involves searching the optimisation space to find the best optimisation configuration that maximises the performance of the input program. However, autotuning usually requires a large number of evaluations to find a good configuration, especially when involving numerous compiler passes. When focusing on optimising execution time, each evaluation requires running the generated binary in an isolated environment, which can be time-consuming.

To address the problem of the high cost of autotuning, predictive model-based techniques have been proposed to accelerate the search process [3, 4]. These techniques rely on offline data collected from a set of training programs. Specifically, each program is compiled and executed under hundreds of randomly generated compiler configurations, each defined by a different combination of optimisation passes, and the performance of each configuration is measured. The collected data – including program features, compiler configurations, and their corresponding performance (e.g., execution time) – is then used to train a machine learning model. This model can either predict the optimal configuration for an unseen program or estimate the performance of configurations to guide the search without profiling. However, the effectiveness of these learning-based methods is often limited by the complexity of the optimisation problem and the availability of high-quality training data. Yet, the application of machine learning-based predictive models to compiler autotuning has remained a largely academic pursuit, with little progress adopted in general-purpose, industrial compilers like LLVM [5]. A small amount of industrial effort has been focused on cases where the optimisation objectives are code size, and only a single pass, such as inlining, is considered [5]. For autotuning complex optimisation tasks, such as compiler phase ordering with a focus on execution time, training a generic, robust, and well-generalised predictive model offline becomes nearly infeasible due to the extensive data requirements for training.

This thesis aims to improve the autotuning efficiency of compiler phase ordering for execution time optimisation. Motivated by offline predictive model-based methods [4], we consider an online model-guided search process without relying on offline training to avoid the limitations of training data requirements. Given that the online model is trained on the fly, it can adapt to unseen programs and hardware platforms. However, online-trained cost models may be inaccurate due to limited data availability. Relying entirely on such models can lead to overfitting to local observations, causing the optimisation process to get stuck in local optima and miss globally better solutions. To address the issue, we employ Bayesian optimisation (BO), which explicitly accounts for model uncertainty and balances exploration and exploitation efficiently. BO's probabilistic framework incorporates uncertainty into the decision-making process, ensuring that the search is not overly reliant on potentially inaccurate predictions. Instead, it explores the optimisation space in a way that balances risk and reward, effectively guiding the search even when model inaccuracies are present. This makes BO a robust choice for optimising compiler phase ordering in dynamic environments, where obtaining precise models is inherently difficult.

Unlocking the potential of BO in compiler auto-tuning requires overcoming several challenges. In particular, the high-dimensional search space of compiler phase ordering poses a significant obstacle. To address this, we propose an optimised BO method with a carefully designed initialisation strategy for general high-dimensional optimisation problems. This method is further customised to leverage the unique characteristics of phase ordering, enabling its effective application to the problem.

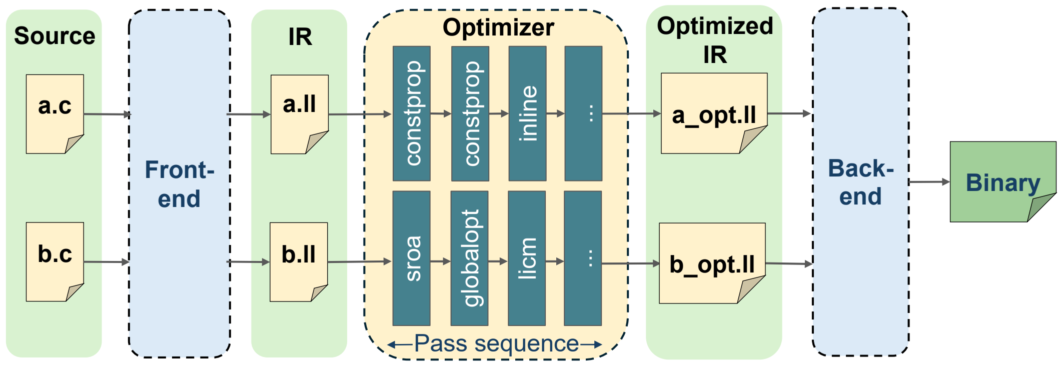


Figure 1.1: The compilation flow for applying customised pass sequences.

The remainder of this chapter discusses the compiler phase ordering problem in detail, highlights the challenges of applying BO, and outlines the key contributions of this thesis. Finally, the overall structure of the document is presented.

1.1 Problem Scope

As depicted in Figure 1.1, modern optimisation compilers are organised as a sequence of three modules: front-end, optimiser (middle-end), and back-end. The front end translates the source code into an intermediate representation (IR), which is then optimised by the optimiser. The optimiser applies a sequence of optimisation passes to the IR to improve the program's performance. Finally, the back-end translates the IR into the target machine code. For programs with multiple source files (e.g., C programs with '.c' files), each file can be treated as an independent optimisation unit, referred to as a *module*.

The optimisation passes used in the middle end can be classified into two categories: analysis and transformation. Analysis passes collect information about the program, such as control flow graphs, data dependence graphs, and alias information, without changing the IR. Transformation passes modify the program to improve performance, such as loop unrolling, inlining, and vectorisation. Transformation passes can be applied in different orders to the IR, and the order in which the passes are applied can significantly affect the performance of the generated code.

In this thesis, we focus on compiler phase ordering, which is the problem of finding the optimal order of optimisation passes to improve the performance of the generated code. Although compiler phase ordering typically does not affect program correctness, rare cases may arise where certain orderings introduce crashes or semantic errors. To ensure correctness, we apply differential testing, a technique that compares the outputs of the original and optimised programs on the same inputs to detect semantic deviations. In our settings, a pass can be applied multiple times within a single pass sequence to optimise an individual source file. This leads to the fact that the search space grows exponentially with the number of optimisation passes and parameters we consider, making it infeasible to perform an exhaustive search for the optimal configuration. Given n optimisation passes, the number of possible sequences of length k is n^k. For example, with 69 passes included in the LLVM 17 -O3 optimisation level and a maximum sequence length of 120, the number of possible pass sequences is 69^{120}, which leads to astronomically large search space. Besides, unlike previous work, we allow the application of different pass sequences to different modules, thus expanding the search space and imposing higher demands on the search algorithm.

This thesis employs Bayesian optimisation (BO) to improve the autotuning efficiency of compiler phase ordering. However, the unique characteristics of the compiler phase ordering problem pose several challenges to the application of BO, which are discussed in the following section.

1.2 Research Challenges

BO provides a principled way to employ an online cost model to improve the efficiency of autotuning the compiler phase ordering problem. However, there are three challenges that must be overcome to realise the goal.

1.2.1 High-Dimensional Optimisation Space

BO is known to be competitive in low-dimensional optimisation problems [6], where it can efficiently balance exploration and exploitation to locate the global optimum. However, its performance deteriorates significantly as the dimensionality or the size of the search space increases. This is particularly problematic for the phase ordering problem, which is inherently high-dimensional. In compiler phase ordering, the search space grows exponentially with the number of available optimisation passes and the maximum sequence length, creating a combinatorial explosion that standard BO techniques struggle to handle.

Despite extensive research on extending BO to high-dimensional problems, the majority of existing methods still exhibit poor performance, particularly in challenging scenarios. Popular approaches like trust region-based local Bayesian optimisation (TuRBO) [7] have shown promise in high-dimensional continuous optimisation by iteratively exploring local regions of the search space. However, they are designed for continuous optimisation problems and are not directly applicable to the phase ordering problem with a complex categorical search space. Moreover, while some recent high-dimensional BO techniques attempt to adapt to categorical or structured spaces, they often assume that the number of categories (the number of passes in the phase ordering problem) is small or rely on additional domain-specific priors to perform well [8, 9]. Neither of these requirements is feasible in the context of compiler autotuning. As a result, the applicability of high-dimensional BO methods to phase ordering tasks remains largely unexplored, leaving a significant gap in optimisation techniques capable of addressing this challenging problem.

1.2.2 Complex Interactions between Compiler Passes

The interactions between different compiler passes are highly complex and vary depending on the programs being optimised, making it extremely difficult to model their combined effects accurately. The non-commutative nature of compiler passes means that one pass's effect can significantly alter subsequent passes' effectiveness. For instance, certain optimisation passes might improve performance in some cases, but if the preceding or subsequent passes are incompatible, the results could be detrimental. As a result, the performance impact of a sequence of passes is not simply the sum of the individual effects but rather is shaped by the complex interdependencies between the passes.

These interactions complicate the task of predicting the performance impact of a given pass sequence, requiring a model that can capture the intricate dependencies between passes. Traditional surrogate models like Gaussian processes [10] used in BO struggle with this level of complexity. Even with more expressive models designed for sequence modelling, such as recurrent neural networks (RNNs) [11] and transformers [12], the high-dimensional and categorical nature of the compiler optimisation search space presents a significant challenge. These models require a large amount of training data to effectively capture the complex relationships within such spaces. However, in practice, the search budget for compiler autotuning is typically limited to only a few hundred evaluations, which is far from sufficient to provide the large dataset needed to train deep learning models like RNNs and transformers without overfitting. Furthermore, incorporating uncertainty estimates into these models would necessitate the use of Bayesian neural networks [13], which are computationally expensive and require careful parameter tuning, making their application even more impractical in this context.

In summary, the combination of complex, non-linear interactions between compiler passes, the high-dimensional and categorical nature of the search space, and the limitations of current machine learning models make it difficult to accurately predict the performance impact of pass sequences.

1.2.3 Practicality Barriers

Although an extensive body of work shows compiler phase ordering can improve application performance [14, 15], the adoption of compiler phase ordering to real-world applications is limited. Firstly, real-world applications are often composed of multiple source files (referred to as *modules* in this thesis). Applying one optimisation sequence to all source files will restrict the optimisation potential, but autotuning all source files one by one is too time-consuming and impractical. Many source files might already be optimised well with a default optimisation sequence like -O3; thus, autotuning them is unnecessary. This requires a multi-module compiler phase ordering approach. Secondly, engineering efforts are required to apply compiler phase ordering to a real-world application. To integrate custom optimisation sequences into the compilation process, users need to understand and then manually re-implement the compilation process of the application as a function to be interacted with by the autotuning algorithms. This is a barrier for users who are not familiar with compiler internals. For real-world applications, the compilation process is often complex and involves many source files, which makes the manual implementation of the compilation process error-prone and time-consuming.

1.3 Contributions

This thesis presents a novel BO-based approach that addresses the challenges outlined above to compiler phase ordering. The proposed approach leverages pass-related compilation statistics to guide the search for the optimal order of optimisation passes. It uses an adaptive BO scheme to dynamically allocate the search budget across multiple source files within a single program. The proposed approach is evaluated on a set of benchmark programs and compared with existing compiler phase ordering approaches. The key contributions of this thesis are:

High-dimensional BO. Firstly, it investigates a largely understudied problem in high-dimensional BO, concerning the impact of acquisition function maximiser initialisation on exploiting acquisition functions' capability. It proposes a simple but effective initialisation strategy to employ multiple heuristic optimisers to leverage the historical data of black-box optimisation to generate initial points for the acquisition function maximiser. Experimental results on a range of heavily studied synthetic functions and real-world applications show that the proposed technique, while simple, can significantly enhance the standard BO and outperform state-of-the-art methods by a large margin in most test cases. In over half of the cases, it achieves better performance than standard BO using only 1/2 of the search budget required for standard BO to converge. The idea of heuristic acquisition function maximiser initialisation can be applied to any type of optimisation problem, including discrete and categorical problems like compiler phase ordering. This addresses the high-dimensional optimisation space challenge (Sec 1.2.1). Chapter 4 presents this work.

Modelling Compiler Pass Interactions. Next, based on the high-dimensional BO method proposed in Chapter 4, it proposes the first BO-based approach for compiler phase ordering by utilising pass-related compilation statistics to model the compiler pass interactions. It also identifies and solves a coverage issue brought by the non-uniform, sparse statistics feature space by customising the acquisition function design. Evaluations conducted on a variety of benchmarks across different platforms demonstrate the approach's superior performance compared to competitive baselines. It achieves comparable tuning results using only 1/3 of the search budget required by other methods. The approach is particularly effective under constrained search budgets: with a budget of 100 runtime measurements, it delivers up to a 17% improvement in program execution speed over random search and up to 10% over the strongest baseline. Especially, on the SPEC CPU 2017 [16] benchmarks, our approach achieves an average improvement of 6% over the default -O3 optimisation level and 2% over random search. These results are particularly meaningful given the size and complexity of the SPEC benchmarks, where achieving even small performance gains is notably difficult. This addresses the challenge of complex compiler pass interactions (Sec 1.2.2). Chapter 5 discusses this work.

Addressing Practicality Barrier. Finally, it presents an adaptive BO scheme to dynamically allocate the search budget across multiple source files within a single program. Experimental results show the adaptive BO scheme can achieve up to a 2.5× faster convergence time. It also develops a user-friendly framework for multi-module compiler phase ordering. The user-friendly framework allows users to autotune the compiler phase order of real-world multi-module applications without rewriting a compilation process. This addresses the practicality barrier challenge (Sec 1.2.3). Chapter 5 presents this work.

1.4 Thesis Outline

The remainder of this thesis is organised as follows.

- Chapter 2 provides the background knowledge of Bayesian optimisation and heuristic optimisation algorithms used in this work.

- Chapter 3 provides a review of the relevant literature. It first reviews prior work on high-dimensional BO and then discusses existing approaches to compiler autotuning. It also outlines the existing work of BO used in code optimisation, before describing prior code characterisation approaches.

- Chapter 4 investigates a largely ignored bottleneck of the high-dimensional BO problem, i.e., the acquisition function maximisation. It proposes a simple but effective strategy to initialise the acquisition function maximiser. By employing multiple heuristic optimisers to leverage the historical data of black-box optimisation to generate initial points for the acquisition function maximiser, it significantly improves the performance of BO in high-dimensional optimisation problems. This chapter is based on the work published in:

  - Jiayu Zhao, Renyu Yang, Shenghao Qiu, and Zheng Wang, "Unleashing the Potential of Acquisition Functions in High-Dimensional Bayesian Optimization", *Transactions on Machine Learning Research (TMLR)*, 2024.

- Chapter 5 develops a novel approach to compiler phase ordering that integrates pass-related compilation statistics into the high-dimensional BO method proposed in Chapter 4. It also presents an adaptive BO scheme to dynamically allocate the search budget across multiple source files within a single program. It evaluates the proposed approach on a diverse set of benchmark programs of varying sizes, highlighting its scalability. This chapter is based on the work published in:

  - Jiayu Zhao, Chunwei Xia, and Zheng Wang, "Leveraging Compilation Statistics for Compiler Phase Ordering," *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2025.

- Chapter 6 concludes the thesis and discusses future research directions.

Chapter 2

Background

This chapter provides an overview of the techniques and theory used in this thesis. Section 2.1 introduces the background of Bayesian optimisation (BO). Section 2.2 discusses heuristic optimisation methods that are employed in the work of this thesis before summarizing this chapter in Section 2.3.

2.1 Bayesian Optimisation

Bayesian optimisation (BO) is a powerful machine-learning-based approach designed to solve expensive black-box function optimisation problems [17], typically formulated as $x^* = \arg\min_x f(x)$. The primary goal of BO is to identify the optimal solution x^* with the minimum number of function evaluations, which is crucial when each evaluation is costly.

BO consists of two main components: a Bayesian statistical surrogate model and an acquisition function (AF). The surrogate model, often implemented as a Gaussian process (GP) [10], provides a probabilistic model of the objective function. This model generates a posterior probability distribution over the possible values of $f(x)$ at any candidate point x, which is updated iteratively as new observations are made. The surrogate model's output includes both the predicted mean and the uncertainty (variance), which are essential for constructing the AF.

The AF guides the search for the optimum by balancing exploration (sampling points with high uncertainty) and exploitation (sampling points with high predicted values). The AF uses the surrogate model's predictions to determine the most promising points to sample next.

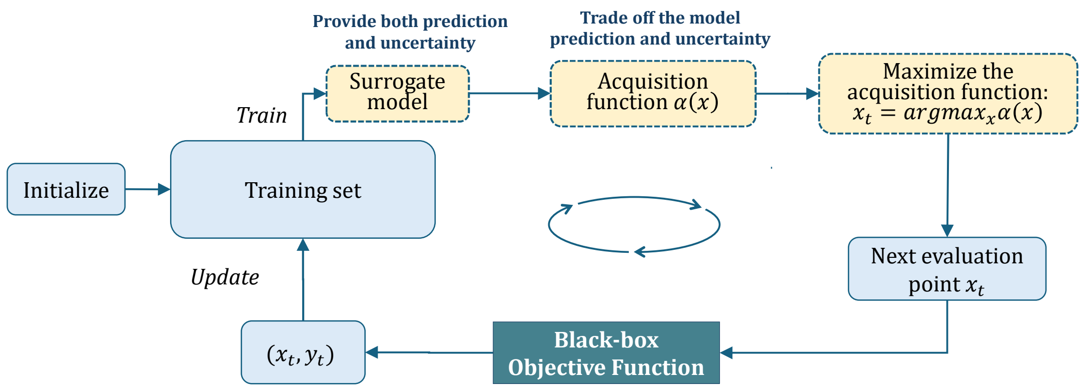

Provide both prediction and uncertainty
Trade off the model prediction and uncertainty
Surrogate model
Acquisition function $\alpha(x)$
Maximize the acquisition function: $x_t = argmax_x \alpha(x)$
Train
Initialize
Training set
Update
Next evaluation point x_t
(x_t, y_t)
Black-box Objective Function

Figure 2.1: The general workflow of Bayesian optimisation.

The general workflow of BO is illustrated in Figure 2.1. The key steps involved in BO are as follows:

1. **Initialisation**: Begin with an initial set of sample points and evaluate the objective function at these points to gather initial data.
2. **Surrogate Model Construction**: Use the initial data to construct the surrogate model, typically a GP, which provides a probabilistic estimate of the objective function.
3. **Acquisition Function Maximization**: Optimise the AF to identify the next candidate point for evaluation. This step involves balancing exploration and exploitation based on the surrogate model's predictions.
4. **Sample Evaluation**: Evaluate the objective function at the candidate point suggested by the AF.
5. **Model Update**: Update the surrogate model with the new data point, refining the posterior distribution of the objective function.
6. **Iteration**: Repeat steps 3 to 5 until a stopping criterion is met, such as a maximum number of iterations or convergence to a satisfactory solution.

In the following two subsections, we introduce Gaussian processes (GPs) and acquisition functions (AFs) in more detail, as they are the core components of BO.

2.1.1 Gaussian Process

Gaussian processes (GPs) [10] have long been the mainstream surrogate model for BO due to their ability to provide a principled framework for modelling uncertainty in function evaluations. To address the computational challenges associated with GPs, alternative approaches such as using random forest [18, 19] and Bayesian neural networks [20, 21] as surrogate models have been proposed, aiming to reduce the computational cost. However, recent advancements in scalable Gaussian processes [22] have significantly alleviated these concerns, making the computational cost of GPs no longer a bottleneck in many applications. Given that GPs continue to demonstrate superior performance across a wide range of tasks, we have also chosen to use GPs in our work to implement BO.

A GP is a collection of random variables, any finite number of which have a joint Gaussian distribution. Formally, a Gaussian process is defined as a distribution over functions, where any finite set of function values evaluated at input points follows a multivariate Gaussian distribution, denoted as:

$$f(x) \sim \mathcal{GP}(m(x), k(x, x'))$$

where $m(x)$ is the mean function and $k(x, x')$ is the covariance function, also known as the kernel function. The mean function $m(x)$ captures the expected value of the function at each input point, while the covariance function $k(x, x')$ represents the similarity between two points x and x', determining how correlated their corresponding function values are in the GP. When used for regression, a GP provides not only the predicted function values $\mu(x)$ but also the uncertainty (variance) $\sigma(x)$ associated with those predictions.

Typically, the mean function $m(x)$ is typically set to a constant 0. Hence, the choice of the kernel $k(x, x')$ defines the properties of the Gaussian process. Commonly used kernels include:

RBF Kernel: The Radial Basis Function (RBF) kernel [23], also known as the squared exponential kernel, is one of the most widely used kernels. It is defined as:

$$k_{\text{RBF}}(x, x') = \sigma_f^2 \exp\left(-\frac{\|x - x'\|^2}{2l^2}\right) \tag{2.1}$$

where σ_f^2 is the signal variance and l is the length scale. The RBF kernel assumes that the function is smooth and varies smoothly over the input space.

Matern Kernel: The Matern kernel is a generalisation of the RBF kernel and provides more flexibility in modelling the smoothness of the function [24]. It is defined as:

$$k_{\text{Matern}}(x, x') = \sigma_f^2 \frac{2^{1-\nu}}{\Gamma(\nu)} \left(\frac{\sqrt{2\nu}\|x - x'\|}{l} \right)^{\nu} K_\nu \left(\frac{\sqrt{2\nu}\|x - x'\|}{l} \right) \tag{2.2}$$

where ν is a parameter that controls the smoothness of the function, $\Gamma(\cdot)$ is the gamma function, and $K_\nu(\cdot)$ is the modified Bessel function of the second kind. Common choices for ν are $\frac{1}{2}$, $\frac{3}{2}$, and $\frac{5}{2}$, which correspond to different levels of smoothness.

These kernels can be combined or extended to create more complex kernels that capture specific properties of the objective function. The choice of kernel depends on the prior knowledge about the function and the specific requirements of the optimisation problem. However, the RBF and Matern kernels are widely used due to their flexibility and effectiveness in modelling a wide range of functions. The length scale parameter l in the kernel function determines the smoothness of the function. A smaller length scale allows the function to vary more rapidly, while a larger length scale enforces smoother variations. In practice, the length scale can differ for each input dimension, allowing the Gaussian process to adapt to varying degrees of smoothness in different directions of the input space. This is achieved by using an anisotropic kernel, where the length scale is a vector $\mathbf{l} = [l_1, l_2, \ldots, l_d]$ with a separate length scale for each dimension d.

For example, the anisotropic RBF kernel is defined as:

$$k_{\text{RBF}}(x, x') = \sigma_f^2 \exp \left(-\frac{1}{2} \sum_{i=1}^{d} \frac{(x_i - x_i')^2}{l_i^2} \right) \tag{2.3}$$

In this formulation, each dimension i has its own length scale l_i, allowing the kernel to capture different levels of smoothness along each dimension. The length scales can be learned from the data by maximising the marginal likelihood of the Gaussian process [25], providing a flexible and powerful way to model complex functions.

2.1.2 Acquisition Function

The acquisition function (AF) is a key component of BO that guides the search for the global optimum. It quantifies the utility of sampling a candidate point x based on the surrogate model's prediction and uncertainty. The goal is to maximise the AF to identify the most promising candidate point for evaluation.

One prevalent AF is the Upper Confidence Bound (UCB) function [26], denoted as:

$$\alpha_{\text{UCB}}(x) = \mu(x) + \sqrt{\beta_t} \cdot \sigma(x) \tag{2.4}$$

where $\mu(x)$ and $\sigma(x)$ are the posterior mean (prediction) and posterior standard deviation (uncertainty) at point x predicted by the surrogate model, and β_t is a hyper-parameter that trades off between exploration and exploitation. A higher value of β_t encourages exploration by giving more weight to the uncertainty term.

Another commonly used AF is the Expected Improvement (EI) [27], which is defined as:

$$\alpha_{\text{EI}}(x) = \mathbb{E}[\max(f(x^+) - f(x), 0)] \tag{2.5}$$

where $f(x^+)$ is the best-observed value of the objective function so far, the EI function quantifies the expected amount of improvement over the current best value. It balances exploration and exploitation by considering the predicted mean and the uncertainty.

The Probability of Improvement (PI) [6] is another AF, defined as:

$$\alpha_{\text{PI}}(x) = \mathbb{P}(f(x) \leq f(x^+)) \tag{2.6}$$

where $f(x^+)$ is the best-observed value of the objective function so far, the PI function measures the probability that the objective function at point x will improve upon the current best value. It focuses more on exploitation by favouring points with high predicted improvement probability.

Each AF has its own advantages and disadvantages. The UCB function is highly adaptable due to its tunable parameters, making it suitable for a variety of optimisation scenarios. The EI function is popular because it effectively balances exploration and exploitation without the need for extensive parameter tuning, which is particularly useful when evaluating the objective function is costly. The PI function is computationally simpler and faster but may risk premature convergence if it does not explore the search space adequately.

In practice, the choice of AF depends on the specific characteristics of the optimisation problem and the desired balance between exploration and exploitation. It is common to experiment with different AFs to determine the most effective one for a given problem.

Monte Carlo Acquisition Function

Monte Carlo acquisition functions (AFs) [28] are introduced here because they offer a flexible and powerful approach to approximating AFs that are otherwise analytically intractable. By leveraging Monte Carlo sampling, these functions can handle complex scenarios, such as batch evaluations and derivative information, which are essential for efficient and effective BO. In the following, we provide details about how Monte Carlo AFs are calculated.

Many common AFs, like the aforementioned UCB EI and PI, can be expressed as the expectation of some real-valued function of the model output(s) at the design point(s):

$$\alpha(X) = \mathbb{E}\big[a(\xi) \mid \xi \sim \mathbb{P}(f(X) \mid \mathcal{D})\big]$$

where $X = (x_1, \ldots, x_q)$, and $\mathbb{P}(f(X) \mid \mathcal{D})$ is the posterior distribution of the function f at X given the data $\mathcal{D}$ observed so far.

Evaluating the AF thus requires evaluating an integral over the posterior distribution. In most cases, this is analytically intractable. In particular, analytic expressions generally do not exist for batch AFs that consider multiple design points jointly (i.e., $q > 1$).

An alternative is to use (quasi-) Monte-Carlo sampling to approximate the integrals. A Monte-Carlo (MC) approximation of α at X using N MC samples is

$$\alpha(X) \approx \frac{1}{N} \sum_{i=1}^{N} a(\xi_i)$$

where $\xi_i \sim \mathbb{P}(f(X) \mid \mathcal{D})$.

For instance, for MC-estimated Expected Improvement, we have:

$$\text{qEI}(X) \approx \frac{1}{N} \sum_{i=1}^{N} \max_{j=1,\ldots,q} \{\max(\xi_{ij} - f^*, 0)\}, \qquad \xi_i \sim \mathbb{P}(f(X) \mid \mathcal{D})$$

where f^* is the best function value observed so far (assuming noiseless observations).

Using the reparameterization trick [29, 30],

$$\text{qEI}(X) \approx \frac{1}{N} \sum_{i=1}^{N} \max_{j=1,\ldots,q} \{\max(\mu(X)_{\_}j + (L(X)\epsilon_i)_{\_}j - f^*, 0)\}, \qquad \epsilon_i \sim \mathcal{N}(0, I)$$

where $\mu(X)$ is the posterior mean of f at X, and $L(X)L(X)^T = \Sigma(X)$ is a root decomposition of the posterior covariance matrix.

The advantages of Monte Carlo AFs include:

- **Batch Support**: They can handle batch evaluations, allowing multiple points to be evaluated simultaneously, which is useful in parallel computing environments.
- **Derivative Information**: They can incorporate derivative information, improving the efficiency and accuracy of the optimisation process.
- **Flexibility**: They can approximate complex integrals that are otherwise intractable, making them suitable for a wide range of AFs and optimisation problems.

Because of these advantages, Monte Carlo AFs rather than analytic AFs are widely used in practice. This thesis also adopts Monte Carlo AFs to handle complex optimisation scenarios and improve the efficiency and effectiveness of BO.

2.2 Heuristic Optimisation

This section describes three popular heuristic optimisation techniques which are used in this thesis: Genetic Algorithms (GAs), Covariance Matrix Adaptation Evolution Strategy (CMA-ES), and Discrete 1+λ Evolution Strategy (DES). GA can be used for both continuous and discrete optimisation problems. CMA-ES only supports numerical continuous optimisation problems. DES is designed for discrete optimisation problems. These methods are employed in Chapters 4 and 5 to achieve high-dimensional BO by better initialising the AF maximisation process.

2.2.1 Genetic Algorithm

Genetic Algorithms (GAs) [31, 32] are a class of optimisation algorithms inspired by the process of natural selection. They are widely used in optimisation problems where the search space is large and complex. We will apply GA in Chapters 4 and 5 to improve the initialisation of the AF maximisation process. GAs maintain a population of candidate solutions and iteratively evolve them to find the optimal solution. The workflow of GAs is shown in Figure 2.2. The key components of GAs are as follows:

1. **Initial Population**: The initial population is generated randomly or based on some heuristic. This population represents a diverse set of potential solutions to the optimisation problem. The size of the population is a crucial parameter that can affect the convergence speed and the quality of the final solution.

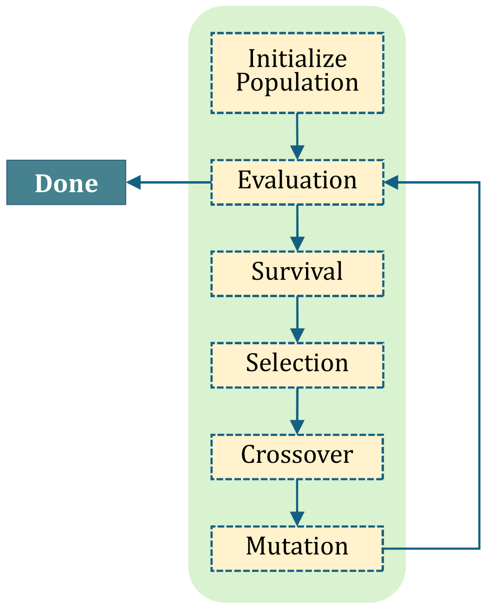


Figure 2.2: The general workflow of a genetic algorithm (GA).

2. **Evaluation**: Each individual in the population is evaluated using a fitness function, which quantifies how good the solution is with respect to the optimisation objective. The fitness function is problem-specific and plays a critical role in guiding the search process.

3. **Survival**: Survival determines which individuals are carried over to the next generation. This can be done using various strategies, such as:

   - **Elitism**: The best individuals are guaranteed to survive to the next generation.
   - **Generational Replacement**: The entire population is replaced by the offspring.
   - **Steady-State Replacement**: Only a few individuals are replaced in each generation.

4. **Selection**: Selection is the process of choosing individuals from the current population to create offspring for the next generation. Common selection methods include:

- **Roulette Wheel Selection**: Individuals are selected based on their fitness proportionate to the total fitness of the population.
- **Tournament Selection**: A subset of individuals is chosen randomly, and the best individual from this subset is selected.
- **Rank Selection**: Individuals are ranked based on their fitness, and selection is based on this ranking.

5. **Crossover**: Crossover, also known as recombination, is the process of combining two parent solutions to produce offspring. Common crossover methods include:

   - **Single-Point Crossover**: A single crossover point is chosen, and the segments of the parents are swapped to create offspring.
   - **Two-Point Crossover**: Two crossover points are chosen, and the segments between these points are swapped.
   - **Uniform Crossover**: Each gene in the offspring is chosen randomly from one of the corresponding genes of the parents.

6. **Mutation**: Mutation introduces random changes to individual solutions to maintain genetic diversity within the population. This helps to avoid premature convergence to local optima. Common mutation methods include:

   - **Bit Flip Mutation**: Each bit in the individual's representation is flipped with a certain probability.
   - **Swap Mutation**: Two genes in the individual's representation are swapped.
   - **Scramble Mutation**: A subset of genes is chosen, and their order is randomly shuffled.

2.2.2 Covariance Matrix Adaptation Evolution Strategy

Covariance Matrix Adaptation Evolution Strategy (CMA-ES) [33, 34] is a popular evolutionary algorithm for continuous optimisation. It is a variant of the Evolution Strategy (ES) family, which is inspired by the natural evolution process. The key idea behind CMA-ES is to maintain a multivariate Gaussian distribution that models the search space. The distribution is iteratively updated to adapt to the landscape of the

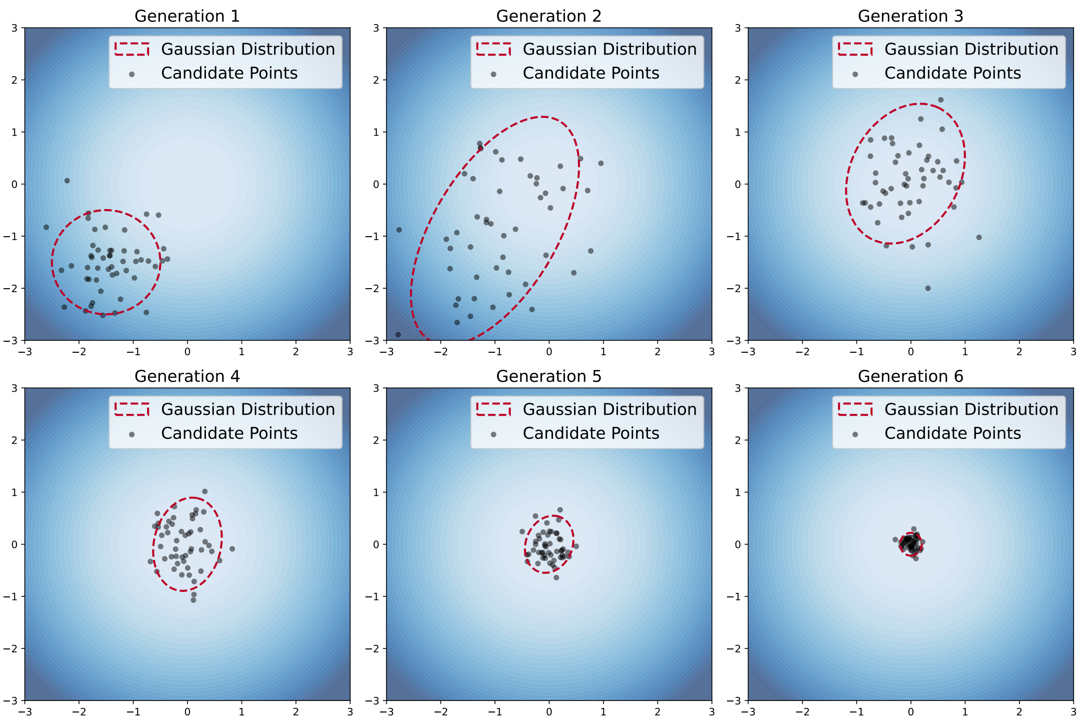


Figure 2.3: Illustration of how CMA-ES evolves on a simple two-dimensional problem. In the plot, the shading indicates the spherical optimisation landscape, where lighter colours represent lower function values, indicating better areas of the optimisation landscape, and darker colours represent higher function values, corresponding to less optimal regions. It clearly shows how the multivariate Gaussian distribution (dashed line) evolves during the optimisation. CMA-ES uses the multivariate Gaussian distribution to generate its candidate evaluation points (dots).

optimisation problem. We will apply CMA-ES in Chapter 4 to improve the initialisation of the AF maximisation process in general continuous BO tasks.

One of the key differences between CMA-ES and standard Evolution Strategies (ES) is that the Gaussian distribution in CMA-ES evolves over time. In standard ES, the search distribution is typically fixed or changes in a predefined manner. However, in CMA-ES, the covariance matrix of the Gaussian distribution is adapted based on the fitness evaluations of the candidate solutions. This allows CMA-ES to learn and exploit the structure of the optimisation landscape, making it more effective at navigating complex, non-linear, and high-dimensional search spaces.

Figure 2.3 illustrates how CMA-ES evolves on a simple two-dimensional problem. The algorithm starts with a multivariate Gaussian distribution that generates candidate evaluation points. The candidate points are evaluated, and the distribution is updated based on the evaluation results. The process is repeated until the algorithm converges to the optimal solution.

The CMA-ES algorithm follows these steps:

1. **Initialisation**: Initialise the mean vector m_0, step-size σ_0, and covariance matrix C_0. Set the generation counter $k = 0$.

2. **Sampling**: Generate λ offspring by sampling from the multivariate normal distribution:
$$x_i \sim \mathcal{N}(m_k, \sigma_k^2 C_k), \quad i = 1, \dots, \lambda \tag{2.7}$$
where λ is the population size.

3. **Evaluation**: Evaluate the fitness of each offspring x_i using the objective function.

4. **Selection**: Select the μ best offspring based on their fitness values. Typically, $\mu \leq \lambda/2$

5. **Adaptation**: Update the mean vector m_k to m_{k+1}, step-size σ_k to σ_{k+1} and covariance matrix C_k to C_{k+1}. Details are show in equations 2.8–2.12.

6. **Termination**: Check the termination criteria (e.g., maximum number of generations, convergence tolerance). If the criteria are met, stop the algorithm; otherwise, increment the generation counter g and repeat from step 2.

Details of the adaptation step are as follows: Firstly, the mean vector is updated using the selected μ offspring:

$$m_{k+1} = \sum_{i=1}^{\mu} w_i \, x_{i:\lambda} \tag{2.8}$$

where $x_{i:\lambda}$ denotes the i-th best offspring (according to their function values) and w_k are predefined weights which satify $w_1 \geq w_2 \geq \cdots \geq w_\mu > 0$. This ensures that the mean vector moves towards the promising regions of the search space. Typically, the weights are chosen such that $\mu_w := 1/\sum_{i=1}^{\mu} w_i^2 \approx \lambda/4$.

The step-size σ_k is updated via path length control, also known as the cumulative step-size adaptation (CSA). The evolution path p_σ is updated as follows:

$$p_\sigma \leftarrow (1 - c_\sigma)p_\sigma + \sqrt{1 - (1 - c_\sigma)^2}\sqrt{\mu_w}\, C_k^{-1/2} \, \frac{m_{k+1} - m_k}{\sigma_k} \tag{2.9}$$

where the update term follows a standard normal distribution $\mathcal{N}(0, I)$ under neutral selection. The step-size then evolves according to:

$$\sigma_{k+1} = \sigma_k \times \exp\left(\frac{c_\sigma}{d_\sigma}\left(\frac{\|p_\sigma\|}{\mathbb{E}\|\mathcal{N}(0, I)\|} - 1\right)\right) \tag{2.10}$$

where key parameters include:

- $c_\sigma^{-1} \approx n/3$, determining the adaptation time scale,
- $\mu_w = \left(\sum_{i=1}^{\mu} w_i^2\right)^{-1}$, representing the variance-effective selection mass,
- $C_k^{-1/2}$ as the symmetric square root of C_k^{-1},
- d_σ, a damping factor typically close to one.

The covariance matrix update follows a similar structure, beginning with the update of the evolution path p_c:

$$p_c \leftarrow (1 - c_c)p_c + \mathbf{1}_{[0,\alpha\sqrt{n}]}(\|p_\sigma\|)\sqrt{1 - (1 - c_c)^2}\sqrt{\mu_w}\frac{m_{k+1} - m_k}{\sigma_k} \tag{2.11}$$

and the covariance matrix is adjusted as:

$$C_{k+1} = (1 - c_1 - c_\mu + c_s)C_k + c_1 p_c p_c^T + c_\mu \sum_{i=1}^{\mu} w_i \frac{x_{i:\lambda} - m_k}{\sigma_k}\left(\frac{x_{i:\lambda} - m_k}{\sigma_k}\right)^T \tag{2.12}$$

where parameters $c_1 \approx 2/n^2$ and $c_\mu \approx \mu_w/n^2$ control the learning rates for rank-one and rank-μ updates, respectively. Other parameters include $c_c = 4/(n + 4)$, $c_s = (1 - \mu_w)c_1 c_c(2 - c_c)$, and $\alpha \approx 2$.

2.2.3 Discrete 1+λ Evolution Strategy

Discrete 1+λ Evolution Strategy (DES) [35–38] is a simple and efficient evolutionary algorithm for discrete optimisation problems. It is based on the Evolution Strategy (ES) framework and is designed to handle combinatorial optimisation problems where the search space consists of discrete variables. Unlike continuous evolution strategies like CMA-ES, which rely on a Gaussian distribution to sample from the search space, DES adopts a discrete mutation sampling strategy instead. We will apply DES in Chapter 5 for compiler phase ordering by generating candidate pass sequences to be evaluated by the AF during BO iterations.

The DES algorithm follows these steps:

1. **Initialisation**: Start with a randomly chosen solution x_0 from the discrete search space.
2. **Mutation-based Sampling**: Generate λ offspring by applying discrete mutations to the current solution x_t. For example, a simple discrete mutation strategy involves randomly selecting a variable/dimension and modifying its value.
3. **Evaluation**: Evaluate the fitness of each offspring using the objective function.
4. **Selection**: Choose the best-performing solution among the parent and the offspring to become the next-generation parent x_{t+1}.
5. **Iteration**: Repeat the mutation-selection cycle until a termination criterion is met.

2.3 Summary

This chapter provides background on optimisation techniques used in this thesis to develop an efficient compiler phase ordering algorithm. The following chapter surveys research literature relevant to this work.

Chapter 3

Related Work

This chapter surveys the literature relevant to this thesis. Section 3.1 reviews the literature on compiler autotuning. Section 3.2 reviews research in Bayesian optimisation (BO), focusing first on high-dimensional BO, then discussing previous strategies for acquisition function (AF) maximisation, as our work shows the AF maximisation process is critical to high-dimensional BO. Section 3.3 surveys the literature on BO in code optimisation. Section 3.4 discusses static IR-level code representation approaches. Finally, Section 3.5 concludes.

3.1 Compiler Autotuning

An extensive body of work shows compiler phase ordering can improve application performance [3]. Compiler autotuning techniques can be broadly categorised into two groups: search-based methods and predictive modelling. Table 3.1 summarises these two categories of compiler autotuning techniques. Search-based methods do not rely on offline training or pre-built models. Instead, they explore the search space dynamically during the optimisation process. Predictive model-based methods rely on offline training to build pre-built models to guide the optimisation process or directly predict the optimal configuration. We review the two categories of compiler autotuning techniques in the following sections.

3.1.1 Search-based Compiler Autotuning

Prior work typically takes a search-based auto-tuning process to navigate the compiler optimisation space. This process involves applying different compiler pass sequences

Table 3.1: Two categories of compiler autotuning techniques.

Category	Characteristic	Literature
Search-based methods	Can adapt to arbitrary autotuning tasks	[14, 39–69]
Predictive Model	Optimising one or a few parameters or focusing on code size reduction	[5, 15, 70–93]

to the target program, profiling the resulting binary, and using the performance measurements as feedback to guide the search. Typical search algorithms, such as random sampling, genetic algorithms (GA), hill climbing and simulated annealing [94], are commonly used to iteratively select the next configuration. The iterative process continues until specific termination criteria are met, such as reaching the maximum number of iterations or exhausting the allocated search time.

Cooper et al. [39, 41] are among the pioneers in the field, and their early work contributes significantly to the literature by exploring the foundational concepts and methodologies. They utilised genetic algorithms (GA) to identify an effective sequence of compiler passes for minimising code size in embedded systems and discovered that GA outperformed random search in finding superior sequences.

Kisuki et al. [40] introduced an iterative compilation framework designed for the autotuning of loop tile sizes and unroll factors, independent of specific architectural constraints, with the objective of minimising program execution times. They assess various iterative strategies, including GA, random sampling, and simulated annealing. Due to the limited scope of their problem settings, the efficiency of these search algorithms demonstrates comparable outcomes.

Almagor et al. [14] conducted an extensive experimental investigation into the compilation sequence space using established benchmarks to assess the compiler phase order space. They employed Genetic Algorithms (GA), hill climbing, and Greedy Constructive Algorithms to navigate this space. Their findings indicate that 80% of local minima within the compilation sequence space are within 5-10% of the optimal solution. By utilising customised search algorithms, they demonstrate that personalised sequences, identified within a range of 200 to 4550 compilation iterations, surpass fixed sequences by 15-25%, underscoring the economic advantages of adaptive compilation methods. Furthermore, they observed that GA frequently yields marginally better results than other techniques. However, these methods rely heavily on exhaustive search and are computationally expensive, making them less practical for large-scale or time-sensitive compilation tasks like compiler phase ordering.

A distinctive characteristic of compiler autotuning is that different compiler configurations can produce identical binaries. This phenomenon is particularly evident in compiler phase ordering. Kulkarni et al. [44] proposed techniques to improve genetic algorithms (GA) for compiler phase ordering by reducing redundant executions. Their approach identifies sequences that generate the same binaries, thereby eliminating unnecessary application runs and enhancing search efficiency. While effective for pruning redundant evaluations, this approach still depends on search heuristics and lacks a cost model to guide the search intelligently.

In addition to single-objective optimisation, Hoste et al. [48] proposed a compiler flag selection framework that utilises multi-objective evolutionary search to automatically discover Pareto-optimal optimisations. By exploring the trade-offs between execution time and compilation time, they analyse the resulting Pareto-optimal flags to provide insights into the significance of different compiler optimisations.

The diversity of real-world programs and the complexity of compiler parameters create a demand for frameworks capable of adapting to varied optimisation landscapes. OpenTuner [62] addresses this challenge by providing ensemble-based search capabilities, which integrate multiple search algorithms including genetic algorithms (GA), hill climbing, simulated annealing, and particle swarm optimisation (PSO) to efficiently navigate the optimisation space. A key feature of OpenTuner is its simultaneous use of ensembles of disparate search techniques. These techniques are dynamically evaluated during the search process, and those demonstrating better performance are allocated a larger proportion of tests. This adaptive approach ensures that the most effective strategies are prioritised, enabling the framework to achieve high-quality optimisation results across diverse programs and parameter configurations.

Similarly, Cummins et al. [67] employed Nevergrad [95], another open-source framework that supports multiple search algorithms, to tackle the compiler phase-ordering problem. Nevergrad adaptively selects the most suitable algorithm based on the problem settings, demonstrating state-of-the-art performance in the CompilerGym environment. Together, these frameworks highlight the importance of adaptive and ensemble-based approaches in addressing the challenges of compiler optimisation. Despite their flexibility, OpenTuner and Nevergrad do not incorporate program-specific or compiler-specific modelling, and lack the ability to capture interactions between compiler phases, which limits their effectiveness in addressing the phase-ordering problem.

Recently, Bayesian optimisation (BO) has also been introduced in compiler autotuning [61]. In Section 3.3, we will discuss in detail how BO has been applied to broader software autotuning problems, such as algorithm configuration tuning. Furthermore, we will highlight the key differences between these existing BO-based approaches and our work, emphasising how our method addresses their limitations and introduces novel contributions to the field.

Overall, pure search techniques are flexible and adaptable, making them suitable for a wide range of compiler autotuning tasks. However, they can be time-consuming and may struggle with high-dimensional search spaces. Predictive model-based techniques presented in the next section, however, use prior knowledge to build a model offline, which either directly predicts the best compilation configuration or guides the search process, thereby significantly reducing the number of executions required for a new program.

3.1.2 Predictive Modelling

Machine learning and data-driven techniques for compiler autotuning have been extensively explored in prior research. These techniques require the collection of offline training data on a set of training programs that sufficiently cover the optimisation space encountered in practice. Each training program is compiled and executed under various compiler configurations, and the performance of each configuration is measured. The resulting data, consisting of program features, compiler configurations, and observed performance, is then used to train the predictive model. Stephenson et al. [70] were pioneers in applying supervised classification to predict loop unroll factors, demonstrating the potential of machine learning in optimising compiler decisions. Their approach relied on manually crafted features. In later studies on loop optimisation, deep learning techniques began to replace handcrafted features, offering improved results. For instance, Ameerhajalicgo et al. [84] developed NeuroVectorizer, an end-to-end framework for predicting vectorisation factors, leveraging deep learning to achieve better performance.

In the area of compiler flag selection, Cavazos et al. [72] proposed a method to leverage performance counters as features to build predictive models to reduce the search space and enhance optimisation efficiency. These performance counters are collected by executing the program compiled with the default -O0 optimisation level and monitoring hardware-level events such as cache misses, branch mispredictions, and instruction counts. The collected counters serve as a dynamic characterisation of the program's behaviour and are used as input features to train machine learning models that predict the most promising compiler optimisations. The MILEPOST GCC framework, introduced by Fursin et al. [75], represented a significant milestone by incorporating machine learning into a self-tuning compiler. This framework automated the selection of compiler optimisations based on program features, paving the way for more adaptive and intelligent compilation systems. More recently, Cereda et al. [15] applied collaborative filtering techniques to leverage knowledge sharing across programs, further improving optimisation outcomes.

Another notable development is the MLGO framework [5], which is the first full integration of ML as a compiler pass in industrial compilers such as LLVM. However, this framework only focused on optimisation objectives like code size, particularly in cases involving a single pass, such as inlining.

There are also attempts to build a predictive model to directly predict the compiler phase order using supervised learning [71, 81, 87] or reinforcement learning [76, 83, 89]. One of the simplest approaches is to predict the optimisation order based on feature-based program similarity. For example, Agakov et al. [71] leverage this similarity together with a nearest-neighbour method to guide the selection of compiler optimisations. However, as collecting sufficient training samples to cover the high-dimensional phase ordering optimisation space is difficult, these approaches either limit the number of compiler passes considered or reduce the search space by grouping compiler phases into sub-sequences. For instance, Kulkarni et al. [76] restricted their approach to considering only up to 7 compiler passes, significantly reducing the complexity of the search space. Similarly, MIComp [81] clustered the passes of the -O3 optimisation level into 5 sub-sequences and focused solely on reordering these sub-sequences. These strategies demonstrate practical compromises to manage the high-dimensional nature of the phase ordering problem, but they also highlight the limitations of current methods in fully exploring the vast optimisation space.

Overall, previous studies underscore the growing importance of data-driven methods in addressing the challenges of modern compiler design and optimisation. However, they often focused on small problems. The application of machine learning-based predictive models to general-purpose, industrial compilers like LLVM remains a problem [5]. Especially, the application of compiler phase ordering for execution time optimisation faces several key challenges that impact its scalability and adaptability. Firstly, the vast diversity of real-world programs makes it difficult to cover all possible scenarios in the training process. Secondly, the optimal pass sequence of programs is highly dependent on the input data, introducing an additional layer of complexity. We can't know in advance the typical workload for an unseen program during training processes. Thirdly, collecting reliable training datasets is expensive and time-consuming for the objective of optimising run time, as it requires running programs in isolated environments to ensure the accuracy of the measurements. Furthermore, the phase ordering search space is astronomically large, leading to extensive training data requirements. Together, these factors make applying offline machine learning to phase ordering to runtime performance optimisation, almost infeasible.

3.2 Bayesian Optimisation

This thesis focuses on high-dimensional Bayesian Optimisation (BO) as a solution to the challenges introduced by the large number of parameters in compiler autotuning tasks. We begin by reviewing existing research on high-dimensional BO and discussing the limitations of applying these methods to the compiler phase ordering problem. Based on our findings in Chapter 4, which highlight the significance of the acquisition function (AF) maximisation process in high-dimensional BO, we subsequently examine prior methods for maximising the AF.

3.2.1 High-Dimensional Bayesian Optimisation

High-dimensional BO has garnered significant attention due to its scalability and practical applicability challenges. These challenges can be broadly categorised into two aspects. First, the number of search iterations required increases with the problem's dimensionality, while the computational cost of Gaussian processes (GPs) grows cubically with the number of search iterations. This makes it difficult to scale BO for high-dimensional tasks. Second, in terms of search efficiency, for most high-dimensional

Table 3.2: High-dimensional Bayesian optimisation techniques.

Technique	**Assumption**	**Literature**
Low-dimensional Embedding	The target function has redundant dimensions	[96–102]
Function Decomposition	The target function has additive structures	[103–108]
Block Coordinate Descent	None	[109–112]
Surrogate Model Re-design	None	[113–115]
Trust Region	None	[7–9, 116]

problems, BO yields worse results than model-free heuristic algorithms with the same number of search iterations (This contrasts with the low-dimensional case, where BO typically outperforms heuristic algorithms).

Fortunately, the first challenge has been largely alleviated by recent developments in scalable GPs [22, 117–122]. Sparse Gaussian Process Regression (SGPR), introduced in [118], reduces the computational cost by approximating the full covariance matrix with a set of inducing variables. This method leverages a variational approach to efficiently model the underlying function. On the other hand, Structured Kernel Interpolation (SKI), proposed in [120], further optimises scalability by interpolating the kernel matrix to reduce memory and computational complexity. Notably, GPyTorch [22], an open-source library for scalable GPs, not only supports the SGPR and SKI methods but uses a modified conjugate gradients algorithm to implement an exact GP inference method with a time complexity of $O(n^2)$, which is significantly faster than the traditional $O(n^3)$ complexity. GPyTorch also provides GPU acceleration and automatic differentiation, making high-dimensional BO more feasible and efficient.

However, the second challenge remains a significant obstacle. To address this issue, researchers have proposed various strategies, as summarised in Table 3.2. These strategies can be broadly categorised into five types: low-dimensional embedding, additive function decomposition, block coordinate descent, surrogate model re-design, and trust region-based local BO. Among these strategies, trust region-based local BO has recently gained popularity due to its simplicity and strong performance. We will discuss these strategies in detail below.

Low-dimensional Embedding

The low-dimensional embedding strategy assumes that the target function has redundant dimensions. By mapping the high-dimensional space to a low-dimensional subspace, standard BO can be performed in this low-dimensional space and then projected back to the original space for function evaluations. This idea was first proposed in REMBO [96] by Wang et al., where the authors introduce the concept of random linear embeddings to reduce the dimensionality of the input space. The random embeddings are implemented by using Gaussian matrix projections, which are supported by theoretical analysis to demonstrate that REMBO preserves the key characteristics of the original function, making it well-suited for high-dimensional black-box optimisation problems. Building upon the REMBO framework, Wang et al. extended the approach to handle even larger dimensions in [97], up to a billion dimensions, making it applicable to large-scale, real-world problems that involve massive search spaces.

While random linear embedding techniques have shown promise in reducing the dimensionality of optimisation problems, their effectiveness has often been limited to cases where the problem exhibits low effective dimensionality, i.e., where only a subset of dimensions influences the objective function. To address this limitation, Qian et al. introduced Sequential Random Embeddings [98], an extension that applies to high-dimensional non-convex problems where all dimensions are effective, but many have only a small bounded effect on the objective function. It refines the embedding space sequentially as the optimisation process progresses, improving the efficiency of optimisation in complex, high-dimensional spaces.

Unlike linear embedding techniques, non-linear embeddings are also used for high-dimensional BO. The work presented in [99] proposed a structured low-dimensional embedding method that leverages the structure of the target function to guide the embedding process. The method uses a variational auto-encoder [123] to capture the underlying function's properties and embeds the input space into a low-dimensional continuous manifold that preserves the neighbourhood relationships of the original function. This approach is particularly effective for functions with complex structures. However, random linear embeddings are more commonly used and favoured compared to non-linear methods due to their simplicity, robustness, and lack of additional modelling requirements.

Then, based on random linear embeddings, Nayebi et al. introduced a theoretically grounded approach called Hashing-enhanced Subspace Bayesian Optimisation (HeSBO) [100]. HeSBO leverages hashing and sketching techniques to perform surrogate modelling and AF optimisation in a low-dimensional subspace. This approach significantly enhances computational efficiency while maintaining strong accuracy guarantees. In contrast to previous methods that relied on Gaussian matrix projections, HeSBO avoids the complications associated with complex corrections in projections.

Binois et al. addressed the challenge of selecting a suitable low-dimensional domain when using linear embeddings for high-dimensional optimisation [101]. They analyse the properties of random embeddings and propose a minimal low-dimensional set that effectively represents the original search space. They also introduce an alternative embedding procedure that simplifies the definition of the low-dimensional domain, improving both practical performance and optimisation efficiency. Their approach is demonstrated to outperform previous methods in BO tasks.

Benjamin et al. identified several crucial issues about the use of random linear embeddings for high-dimensional BO and proposed Adaptive Linear Embedding for Bayesian Optimisation (ALEBO) [102]. ALEBO improves high-dimensional BO by using adaptive linear embeddings with Mahalanobis kernels and polytope constraints. These innovations enhance the model's ability to capture function structure, maintain feasible search regions, and ensure high probability containment of the global optimum.

Overall, random embeddings, while supported by theoretical guarantees, still require the target function to have redundant dimensions in order to perform well. However, since the effective dimensionality is unknown for black-box functions, selecting the optimal embedding dimension remains a challenge. Even state-of-the-art methods like ALEBO and HeSBO still lag behind model-free evolutionary algorithms in high-dimensional real-world tasks [102]. They are also limited to optimisation tasks with continuous search space, making them unsuitable for compiler autotuning.

Function Decomposition

Function decomposition methods assume that the target function has an additive structure, i.e., variables in the design space are separable. In that case, the target black-box function f can be decomposed into the following additive form,

$$f(x) = f^{(1)}(x^{(1)}) + f^{(2)}(x^{(2)}) + \cdots + f^{(M)}(x^{(M)})$$

where each $x^{(j)}$ represents a lower-dimensional component that includes several related variables. This property can be exploited to simplify the optimisation process. By decomposing the high-dimensional function into a sum of lower-dimensional functions, the effective dimensionality of the model is the largest dimension among all additive groups $x^{(1)}, x^{(2)}, ..., x^{(M)}$, thus reducing the complexity of the optimisation task. One of the earliest works in this area is Add-GP-UCB [103], which models the target function as a sum of GPs, each defined over a subset of dimensions. This approach significantly reduces the search complexity and improves scalability.

Building on the Add-GP-UCB framework, Wang et al. proposed a batched version of additive BO [104], which allows for parallel evaluations of the target function. This method leverages the additive structure to efficiently allocate computational resources, making it suitable for large-scale optimisation tasks. An enhanced version of this approach is Ensemble Bayesian Optimisation (EBO) [108]. Unlike [104] which only learns the additive structure, EBO jointly learns both the additive structure and the kernel parameters.

In practical applications, the additive structure is not always known a priori. Thus, Gardner et al. introduced a method for discovering additive structures in high-dimensional functions [105]. Their approach uses a greedy algorithm to identify subsets of dimensions that contribute additively to the target function. By iteratively selecting and combining dimensions, the method efficiently decomposes the function into additive components, improving the optimisation process.

Rolland et al. [106] extended the additive model approach by considering additive models with arbitrary overlap among the subsets of variables. This generalisation lifts the assumption that the subsets are disjoint, allowing for more flexible modelling of the target function. By representing the dependencies via a graph, they deduced an efficient message passing algorithm and provided an algorithm for learning the graph from samples based on Gibbs sampling. Mutny et al. [107] also proposed a generalised additive model with possibly overlapping variable groups. For non-overlapping groups, they provide the first provably no-regret polynomial time algorithm. They introduce a novel deterministic Fourier Features approximation for the squared exponential kernel, reducing the kernel matrix inversion complexity from cubic to essentially linear.

While additive function decomposition methods have shown great promise in many cases, their performance is highly dependent on the presence of additive structures in the target function, which may not always be present in real-world problems like compiler autotuning. Besides, even for high-dimensional functions with additive structures, the state-of-the-art additive method EBO failed to outperform model-free evolutionary algorithms [7, 108]. This is actually because EBO uses random search to maximise the AF. In Chapter 4, we will demonstrate the critical role that the AF maximisation process plays in the performance of high-dimensional BO.

Block Coordinate Descent

Block coordinate descent (BCD) is an optimisation method that iteratively optimises a subset of variables (called a 'block') while keeping the others fixed [124]. In the context of Bayesian optimisation (BO), BCD can be combined with BO to improve the efficiency of the optimisation process. The main idea is to decompose the high-dimensional optimisation problem into a series of lower-dimensional sub-problems, each of which can be solved using BO. By iteratively optimising these sub-problems, BCD-based BO can effectively navigate the high-dimensional search space.

One of the earliest works that combined BCD with BO is Dropout BO [109], which randomly selects a subset of dimensions to optimise at each iteration. Another method, LineBO [110], optimises one dimension at a time by performing a line search along each dimension using BO. CobBO [112] extends these methods by introducing a two-stage approach. It first uses a computationally efficient kernel for global exploration and then switches to a more sophisticated kernel for local optimisation in selected subspaces. Additionally, CobBO reduces computational costs by introducing virtual points and dynamically selecting promising subspaces, effectively balancing exploration and exploitation. These methods are simple and easy to implement, but they still underperform model-free evolutionary algorithms in most real-world high-dimensional optimisation tasks. For compiler phase ordering, the effect of one optimisation pass often depends on the others, meaning that optimising a subset of dimensions in isolation may not capture the critical interdependencies.

Surrogate Model Re-design

In addition to the aforementioned techniques that aim to reduce the problem dimensionality, some methods focus on redesigning the surrogate model to enhance the performance of high-dimensional BO. These methods are based on the idea that the failure of high-dimensional BO is often due to the surrogate model's inability to capture the complex structure of the target function. Note this idea is not always the case, as our work in Chapter 4 will show that inadequate AF maximisation often leads to the poor performance of high-dimensional BO.

Rana et al. proposed the Elastic Gaussian Process (EGP) [113], which adapts the Gaussian process model to high-dimensional spaces by introducing an elastic squared exponential kernel to dynamically adjust its length-scales from large to small. Oh et al. found that high-dimensional BO always spends too many evaluations near the boundary of its search space. They introduced Cylindrical Kernels [114], which use a cylindrical transformation to transform the ball geometry of the search space to contract the volume near the boundaries.

Sparse Axis-Aligned Subspaces Bayesian Optimisation (SAASBO) [115] is a most recent method that addresses the challenge of high-dimensional optimisation by identifying and exploiting sparse, axis-aligned subspaces. SAASBO uses a sparsity-inducing prior for automatically identifying and selecting significant dimensions. This method has shown strong performance in high-dimensional tasks, particularly when the underlying function depends on only a few important dimensions. However, for problems with non-axis-aligned structures like compiler autotuning, the performance improvement is not significant.

Trust region Based Local Bayesian Optimisation

Trust region-based local Bayesian optimisation (TuRBO) [7] is a prominent method that addresses the challenges of high-dimensional BO motivated by the success of trust region methods. TuRBO maintains a scalable trust region and performs local Bayesian optimisation within this region. The size of the trust region is dynamically adjusted based on the optimisation process: if multiple consecutive searches fail to improve the objective, the trust region is halved in all dimensions. This process allows TuRBO to gradually focus on finer regions, identifying sparse high-performance points in high-dimensional spaces.

Building on the success of TuRBO, several extensions and variations have been proposed. Wang et al. introduced LaMCTS [116], which integrates Monte Carlo Tree Search (MCTS) with local BO. This method leverages the hierarchical structure of MCTS to dynamically partition the search space and apply BO in the partitioned subspace. However, if only using standard BO in its subspaces, this method does not outperform TuRBO in practice.

Wan et al. proposed a method that extends TuRBO to high-dimensional binary and continuous mixed search spaces [8]. Their approach defines a customised GP kernel to deal with the binary or mixed input variables. They also provide a trust region construction on binary search space. Their strategy enhances the efficiency and effectiveness of BO in complex search spaces. However, this method cannot be applied to the compiler phase-ordering problem, which involves a high-dimensional categorical search space. In the compiler phase-ordering problem, each dimension can have more than 70 possible categorical values, making it challenging for the proposed kernel to effectively capture the complex structure of the target function.

Maus et al. applied TuRBO to the field of drug discovery [9], which relies on Variational Autoencoders (VAE) operating in a latent space learned from structured inputs. By optimising in the latent space, Local Latent Space Bayesian Optimisation (LLSBO) can capture the underlying structure of the input space, leading to more efficient optimisation in high-dimensional tasks with structured inputs.

These advancements demonstrate the potential of trust region-based local BO methods in addressing the challenges of high-dimensional optimisation. While trust region-based local BO methods become increasingly popular, they attribute the success of trust region-based BO algorithms to the local modelling alleviating the surrogate model's inability in the global space. However, this idea is not verified. Motivated by the success of TuRBO, we try to explore the reasons for the poor performance of high-dimensional BO. We note TuRBO not only explicitly changes the global surrogate modelling to local modelling, but also implicitly limits the search space of the AF maximisation. This work thus investigates whether the poor performance of high-dimensional BO is also due to the inadequate AF maximisation process. As we will show in Chapter 4, by simply changing the initialisation of the AF maximiser, the global surrogate model which is used in standard BO can outperform TuRBO. In the next section, we will review the literature on AF maximisation.

3.2.2 Acquisition Function Maximization

Given the posterior belief, BO constructs an AF and maximises it to select new queries. Table 3.3 summarises the AF maximisation techniques used in previous BO literature.

Table 3.3: Acquisition function maximisation techniques used in previous BO literature.

Technique	Details	Literature
Random Search	• Simplest method, effective for low-dimensional problems	[104, 107, 108, 110, 125–128]
Evolutionary Algorithms	• Random initialization by randomly generating initial population • Mainly based on GA and CMA-ES	[96, 97, 100, 103, 129–131]
Multi-random-start Local Search	• Random initialisation by selecting multiple restart points from a set of randomly generated points • Local search can be both derivative-free and derivative-based	[18, 25, 28, 102, 112, 114, 115, 128, 132–138]
Top-n Local Search	• Non-random initialisation by using the best-observed points in previous target black-box function optimisation history	[19, 139]

There are four mainstream AF maximisation techniques: random search, evolutionary algorithms, multi-random-start local search and top-n local search.

Random search is efficient in low-dimensional problems [125–127] but can be inadequate for high-dimensional problems [18]. Evolutionary algorithms are often used where gradient information is unavailable [130, 131]. For AFs that support gradient information, a gradient-based optimisation method might be a better choice. However, given that AFs are generally non-convex and often flat, a gradient-based optimisation method can be trapped in local optima. To address this issue, a multi-random-start strategy is used to generate multiple initial restart points for gradient-based optimisation and shows better performance in AF maximisation than evolutionary algorithms [28]. Note the multi-random-start strategy can also be combined with derivative-free local optimisation methods such as a local Gaussian sampling [18], which also achieves better AF maximisation results than using a single-start evolutionary algorithm.

We note that the aforementioned AF maximisation techniques use randomly generated raw samples to initialise their AF maximisers. GPyOpt [139] and Spearmint [19] are two of the few works that provide different initialisation strategies from random initialisation. Instead, they rely on the top n best-observed points in the previous black-box optimisation iterations to initialise the AF maximiser. GPyOpt combines random points with the best-observed points to serve as the initial points. Spearmint uses a Gaussian spray around the incumbent best to generate initial points. However, no empirical study shows that these initialisation strategies are better than random initialisation. As such, random initialisation remains the most popular strategy for high-dimensional BO AF maximisation.

As the dimensionality increases, even the state-of-the-art multi-start gradient-based AF maximiser struggles to globally optimise the AFs. In such cases, the initialisation of the AF maximiser greatly influences the quality of AF optimisation. The initialisation phase of AF maximisers is often overlooked. Chapter 4 will delve into this underexplored problem.

3.3 Bayesian Optimisation in Code Optimisation

Some works have employed BO for software tuning (including not only compiler autotuning but also the autotuning of other software parameters). We summarised these works in Table 3.4. These include Flash [142], BOCA [61], Bliss [141], Ytopt [140], and

Table 3.4: Bayesian Optimisation In Software Optimisation.

Literature	Domain	#Dimensions	Space Size
Ytopt [140]	Loop Optimisation Parameters	5 – 10	$10^4 - 10^5$
Bliss [141]	Parallel Application Configurations	5 – 9	$10^4 - 10^8$
Flash [142]	Software System Configurations	3 – 17	$10^2 - 10^5$
BaCO [143]	Kernel Optimisation	4 – 15	$10^2 - 10^{11}$
BOCA [61]	Compiler Flag Selection	64 – 71	$10^{19} - 10^{21}$
Our work [144]	Compiler Phase Ordering	120	10^{225}

BaCO [143]. Flash uses decision trees to build its surrogate model to autotune configurations of various software systems. BOCA uses the random forest as its surrogate model for the compiler flag selection problem. Bliss utilises an ensemble of diverse GP models and AFs to autotune parallel applications. Ytopt uses the traditional Skopt [128] BO library to optimise LLVM Clang/Polly pragma configurations on the PolyBench benchmark suite. BaCO customises its BO implementation to support different parameter types and constraints for kernel optimisation on 15 important kernels from domains like machine learning, statistics, and signal processing.

These frameworks are mainly used for tuning a small number of parameters in the code optimisation domain. The only exception is BOCA, which optimises up to 71 parameters. However, BOCA only considers the compiler flag selection problem, which involves a binary decision space, making the search space much smaller than the compiler phase-ordering problem. In contrast, this thesis considers a larger and more realistic optimisation space for compiler phase ordering, involving 76 distinct passes and pass sequences of length 120. This design is inspired by the structure of the -O3 optimisation level in LLVM 17, which includes 72 passes and a pass sequence of length 96. Besides, existing frameworks also use the original tuning parameters as the input to fit their surrogate models, which is the same as the standard BO method.

While these frameworks are effective for their respective tasks, they do not apply to the compiler phase-ordering problem for two main reasons: (1) the high-dimensional nature of the search space, and (2) the lack of effective surrogate models to capture the complex interactions between compiler phases. In Chapter 5, our work addresses these challenges by applying the high-dimensional BO method introduced in Chapter 4 and utilising pass-related compilation statistics to more effectively capture the interactions between compiler phases. One key difference between our work and previous BO-based code optimisation approaches is that we use post-compilation statistics as features to fit the surrogate model. In the next section, we will review the literature on code characterisation to provide further context.

3.4 Code Characterization

Standard BO uses original tuning parameters as the input to fit the surrogate model. However, for the compiler phase ordering problem, the original tuning parameters may not be the best features to capture the complex interactions between compiler phases. The post-compilation information could provide more information about the interactions between compiler phases. In this section, we review the literature on code characterisation which could be used to extract features from the intermediate representations (IRs). We focus solely on static code features because our goal is to use these features to build a cost model that predicts the performance of a given compilation sequence. Dynamic code features, which require profiling, are unsuitable for our online autotuning scenario.

Table 3.5 summarises the relevant techniques. DeepTune-IR [145] represents the IR of OpenCL programs using token sequences. This method leverages tokenisation to capture the program's semantics in a sequence-based manner, which can be useful for tasks such as program optimisation and tuning based on sequence patterns. Autophase [146] develops analysis passes to extract statistical static features from the IRs, such as the number of branches and the number of add instructions. These features provide valuable insights into the program's structure and behaviour, helping to identify potential optimisation opportunities such as excessive branching. However, neither DeepTune-IR's sequence-based features nor Autophase's statistical features can capture the underlying graph structure of the code, which is crucial for representing dependencies within the program. This limitation can hinder the model's ability to fully understand the relationships between different program elements.

To better capture the structured nature of programs, syntactic (tree-based) as well as semantic (graph-based) representations have been proposed. CDFG [147] uses graph neural networks (GNNs) to represent the intermediate representation (IR) of OpenCL programs based on abstract syntax trees (ASTs) and control flow graphs (CFGs). This approach models the control flow structure of OpenCL programs, making it suitable

Table 3.5: Code Characterization.

	Source Languages	**Representation**	**Flow-sensitive?**	**Position-sensitive?**	**Value-sensitive?**
CDFG [147]	OpenCL	IR graph	✓		
DeepTune-IR [145]	OpenCL	IR token Sequence		✓	
inst2vec [148]	C++, OpenCL	IR statement embedding	✓		✓
Autophase [146]	C, C++, Fortran, Haskell, OpenCL, Swift	IR statistical features			
IR2vec [149]	C, C++, Fortran, Haskell, OpenCL, Swift	IR embedding	✓		
Programl [150]	C, C++, Fortran, Haskell, OpenCL, Swift	IR graph	✓	✓	✓

for analysing control dependencies and execution paths within the program. Inst2vec [148] combines control flow with dataflow in order to learn unsupervised embeddings of LLVM-IR statements. IR2vec [149] combines representation learning methods with flow information to directly represent IRs as distributed embeddings in continuous space. Programl [150] represents programs as graphs with explicit control, data, and call edge types. Programl captures a greater range of intra-program relations than prior graph representations to accurately capture the semantics of programs. It also employs pre-trained inst2vec embeddings to obtain continuous embedding vectors.

These prior feature extraction methods have shown promise in various compiler tasks. However, they are primarily designed to focus on characterising different programs, and they struggle to detect the changes in a single program caused by different compiler passes. For example, the *function-attrs* pass could significantly affect the performance of some programs, but its transformation on the program cannot be recognised by the above-mentioned code characterisation techniques. This is because *function-attrs* only changes the function attributes which are not considered in those techniques.

3.5 Summary

This chapter describes the relevant literature in the fields of high-dimensional BO, compiler autotuning, recent applications of BO to code optimisation, and IR-based code characterisation. We have identified the challenges of high-dimensional BO and compiler phase ordering and reviewed the existing techniques that have been proposed to address these challenges. We have also discussed the limitations of existing methods and highlighted the gaps in the literature that our work aims to address. The next chapter will present our proposed approach to address the challenges of high-dimensional BO.

Chapter 4

Understanding Challenges of High-Dimensional Bayesian Optimisation

This chapter presents an empirical study on the initialisation of the acquisition function (AF) maximiser and proposes a high-dimensional Bayesian optimisation (BO) method, which is a crucial step towards applying BO to the high-dimensional compiler phase ordering problem. Phase ordering involves complex interactions between compiler passes, and thus direct application of BO requires further customisation, which is addressed in Chapter 5. This chapter instead focuses on the high-dimensional BO problem itself, demonstrating our method's effectiveness on generic optimisation tasks and validating it in compiler autotuning scenarios such as compiler flag selection. It is organised as follows: Section 4.2 motivates the importance of AF maximisation in high-dimensional BO. Section 4.3 describes the methodology of the study. Section 4.4 presents the experimental setup. Section 4.5 presents and discusses the experimental results. Finally, Section 4.6 concludes the chapter.

4.1 Introduction

As a black-box optimisation technique, Bayesian optimisation (BO) has demonstrated promising results in many fields, such as hyper-parameter tuning [151], material design [152], and robotic navigation [153, 154]. However, while BO performs well for low-dimensional problems, its effectiveness in high-dimensional settings often lags behind other techniques [155]. This limits its applicability to high-dimensional compiler autotuning tasks.

To better understand the challenges of high-dimensional BO, it's essential to re-examine the process of BO. As shown in Figure 4.1, BO builds an online probabilistic surrogate model, typically a Gaussian process [24], to guide the search. This surrogate model defines an acquisition function (AF) that balances exploitation (model prediction) and exploration (model uncertainty). The next query point is determined by maximising the AF, which helps identify promising candidates for evaluation. Hence, the performance of BO hinges on both the model-based AF and the process of maximising this AF, with either potentially becoming a bottleneck in high-dimensional BO. Most prior research has concentrated on the former, focusing on the design of surrogate models and AFs [19, 26, 114, 156–159], with limited attention given to improving the latter.

Recent advancements in AF maximisation have introduced a multi-start gradient-based AF maximiser for batch BO scenarios, which has shown superior results compared to random sampling and evolutionary algorithms [28]. However, as the dimensionality increases, even this advanced AF maximiser faces challenges in globally optimising the AF. In such high-dimensional cases, the initialisation of the AF maximiser significantly affects the quality of AF optimisation. Yet, the impact of AF maximiser initialisation on the realisation of AF's potential and the overall BO performance remains unclear. Our examination of state-of-the-art BO packages reveals that random initialisation (selecting initial points from a set of random points) or its variants are commonly used as default strategies for AF maximiser initialisation. This is the case for widely-used BO packages like BoTorch [25], Skopt [128], Trieste [132], Dragonfly [130] and GPflowOpt [133]. Specifically, GPflowOpt, Trieste and Dragonfly select n points with the highest AF values from a set of random points to serve as the initial points. BoTorch uses a similar but more exploratory strategy by performing Boltzmann sampling instead of relying solely on AF values to select from the random candidates. Likewise, Skopt directly selects initial points by uniformly sampling from the global search domain. Furthermore, various high-dimensional BO implementations adopt random initialisation as their default strategy [102–104, 108, 114, 156]. GPyOpt [139] and Spearmint [19] are two of the few works that provide different initialisation strategies from random initialisation. GPyOpt combines random points with the best-observed points to serve as the initial points. Spearmint uses a Gaussian spray around the incumbent best

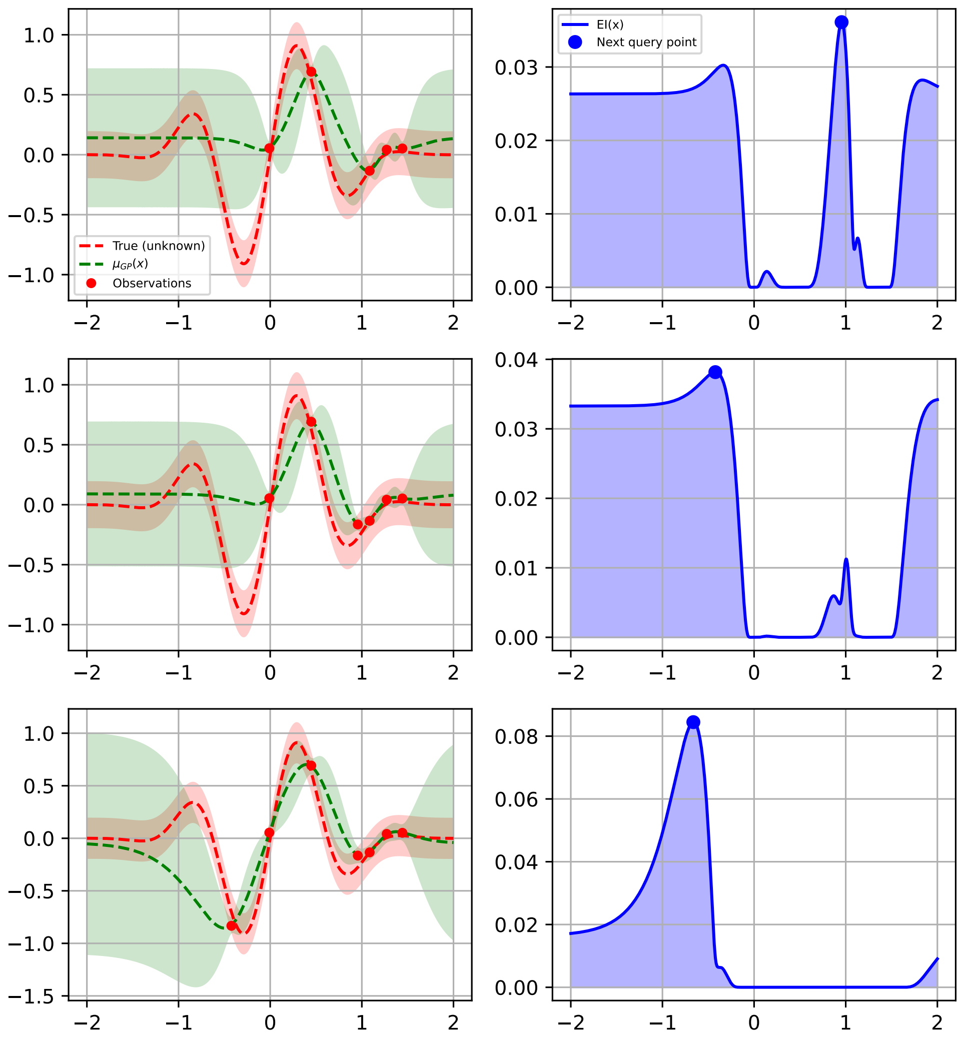


Figure 4.1: Illustration of Bayesian optimisation with an EI acquisition function, where the rows from top to bottom correspond to increasing optimisation iterations. The left panel shows the optimisation process: red dots indicate observed data points (evaluated configurations), the red dashed curve represents the true (unknown) objective function, and the green dashed curve with shaded area depicts the surrogate model prediction with its confidence interval. The right panel shows the corresponding AF (EI), where the blue dot marks the next evaluation point suggested by the AF.

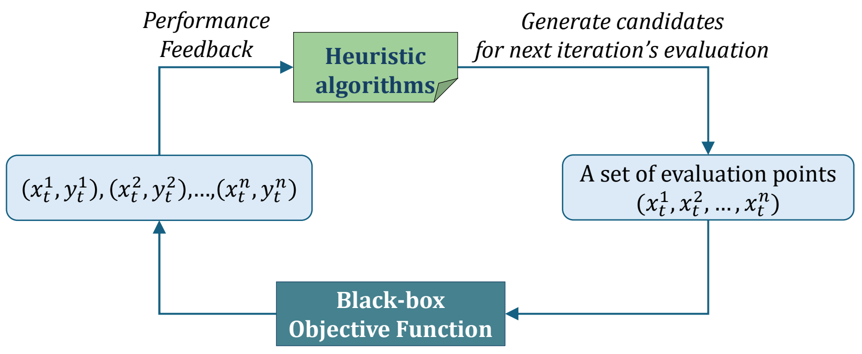
to generate initial points. However, no empirical study shows that these initialisation strategies are better than random initialisation. As such, random initialisation remains the most popular strategy for high-dimensional BO AF maximisation. There is a need to understand the role of the initialisation phase in the AF maximisation process.

This chapter systematically investigates the impact of AF maximiser initialisation on high-dimensional BO, with respect to AF optimisation quality and its impact on final search performance. This is motivated by an observation that the pool of available candidates generated during AF maximisation often limits the AF's power when employing the widely used random initialisation for the AF maximiser. This limitation asks for a better strategy for AF maximiser initialisation. To this end, our work provides a simple yet effective AF maximiser initialisation method to unleash the potential of an AF. Our key insight is that when the AF is effective, the historical data of black-box optimisation could help identify areas that exhibit better black-box function values and higher AF values than those obtained through random searches of AF.

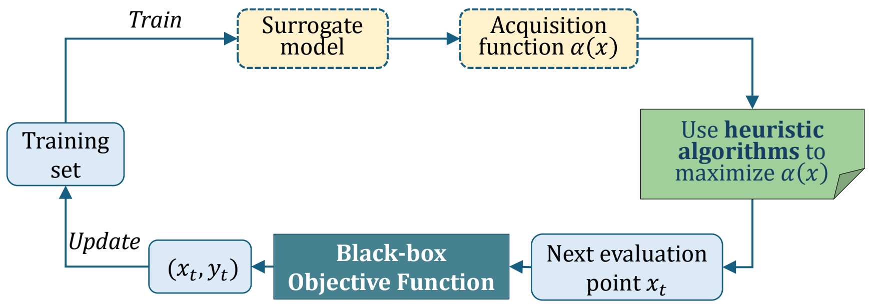
We develop AIBO[1], a Python framework to employ multiple heuristic optimisers, like the covariance matrix adaptation evolution strategy (CMA-ES) [33] and genetic algorithms (GA) [160], to utilise the historical data of black-box optimisation to generate initial points for a further AF maximiser. We stress that the heuristics employed by AIBO are not used to optimise the AF. Instead, they capitalise on the knowledge acquired from the already evaluated samples to provide initial points to help an AF maximiser find candidate points with higher AF values. For instance, CMA-ES generates candidates from a multivariate normal distribution determined by the historical data of black-box optimisation. To investigate whether performance gains come from better AF maximisation, AIBO also incorporates random initialisation for comparison. Each BO iteration runs multiple AF initialisation strategies, including random initialisation on the AF maximiser, to generate multiple candidate samples. It then selects the sample with the maximal AF value for black-box evaluation. Thus, heuristic initialisation strategies work only when they identify higher AF values than random initialisation.

To avoid confusion, we compare in Figure 4.2 the heuristic algorithms in AIBO with two other common use cases: one where the heuristic algorithms are applied directly to optimise the target black-box function, and another where they are applied to directly

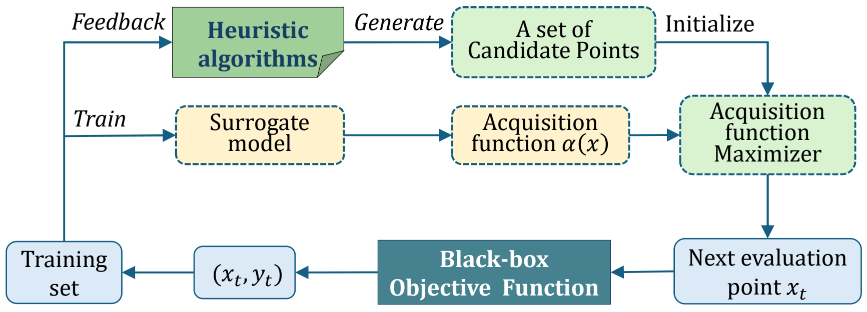
[1] AIBO =Acquisition function maximiser Initialization for Bayesian Optimization.


Performance Feedback
Heuristic algorithms
Generate candidates for next iteration's evaluation
$(x_t^1, y_t^1), (x_t^2, y_t^2), \ldots, (x_t^n, y_t^n)$
A set of evaluation points $(x_t^1, x_t^2, \ldots, x_t^n)$
Black-box Objective Function

(a) Use heuristic algorithms to optimise the target black-box function.


Train
Surrogate model
Acquisition function $\alpha(x)$
Training set
Use heuristic algorithms to maximize $\alpha(x)$
Update
(x_t, y_t)
Black-box Objective Function
Next evaluation point x_t

(b) Use heuristic algorithms to optimise the acquisition function in BO.


Feedback
Heuristic algorithms
Generate
A set of Candidate Points
Initialize
Train
Surrogate model
Acquisition function $\alpha(x)$
Acquisition function Maximizer
Training set
(x_t, y_t)
Black-box Objective Function
Next evaluation point x_t

(c) Heuristic algorithms in AIBO.

Figure 4.2: Comparison of heuristic algorithms in AIBO and other optimisation scenarios. Here, x_t^i denotes the i-th candidate point generated at iteration t, which corresponds to a compiler configuration in the context of compiler autotuning. The associated y_t^i represents its measured objective value (e.g., program runtime).

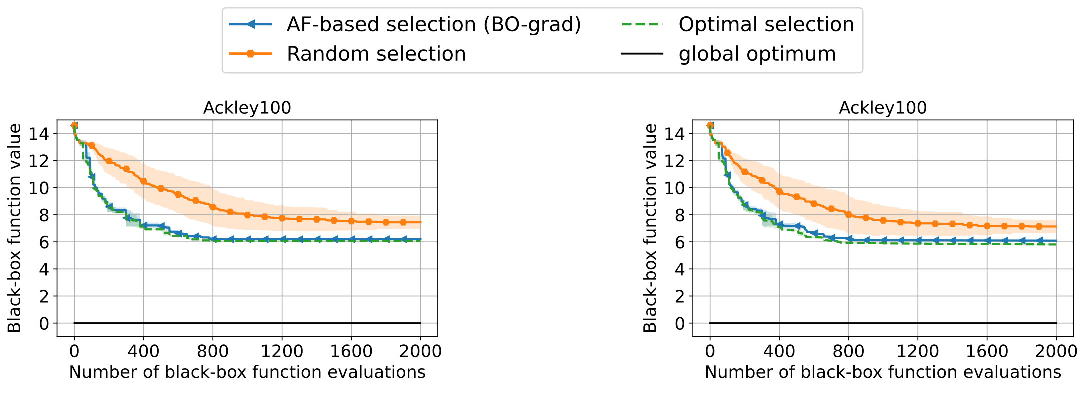
optimise the acquisition function. Note that optimising acquisition functions is usually more difficult than optimising the target black-box function; recent work has shown that commonly used acquisition functions often face numerical pathologies [137]. This is why evolutionary algorithms might succeed in optimising the target function but fail to effectively optimise the acquisition functions.

To demonstrate the benefit of AIBO, we integrate it with the multi-start gradient-based AF maximiser and apply the integrated system to synthetic test functions and real-world applications with a search dimensionality ranging between 14 and 300. Specifically, we set up a compiler flag selection task to examine whether the proposed method is applicable to the field of compiler autotuning. Experimental results show that AIBO significantly improves the standard BO not only in synthetic functions but also in real-world tasks including compiler autotuning. Our analysis suggests that the performance improvement comes from better AF maximisation, highlighting the importance of AF maximiser initialisation in unlocking the potential of AF for high-dimensional BO.

The contribution of this chapter is two-fold. Firstly, it investigates a largely ignored yet significant problem in high-dimensional BO concerning the impact of the initialisation of the AF maximiser on the realisation of the AF capability. It empirically shows that the commonly used random initialisation strategy limits AFs' power, leading to over-exploration and poor high-dimensional BO performance. Secondly, it proposes a simple yet effective initialisation method for maximising the AF, significantly improving the performance of high-dimensional BO. Benefiting from the simplicity of the proposed method, it can be applied to various optimisation scenarios, laying the foundation for addressing high-dimensional issues in compiler phase ordering.

4.2 Motivation

This section demonstrates that the typical randomly initialised acquisition function (AF) maximisation process is inadequate for high-dimensional BO. It also highlights that this issue exists in the compiler autotuning domain.



(a) 10 AF maximiser restarts

(b) 1000 AF maximiser restarts

Figure 4.3: Evaluating the sample chosen by AF-based selection against random and optimal selection among all intermediate candidate points generated during the AF maximisation process when applying BO-grad to 100D Ackley functions. We use two random initialisation settings for AF maximisation: **(a)** 10 restarts and **(b)** 1000 restarts. In both settings, the performance of the native BO-grad (AF-based selection) is close to optimal selection and better than random selection, suggesting that the AF is effective at selecting a good sample from all candidates but is restricted by the pool of available candidates. Increasing the number of restarts from 10 to 1000 does not enhance the quality of intermediate candidates, indicating that a more effective initialisation scheme, as opposed to random initialisation, is necessary.

4.2.1 A Synthetic Function as Motivation

As a motivation example, consider applying BO to optimise a 100-dimensional black-box Ackley function [161] that is extensively used for testing optimisation algorithms. The goal is to find a set of input variables (x) to minimise the output, $f(x_1, \ldots, x_{100})$. The search domain is $-5 \leq x_i \leq 10, i = 1, 2, \ldots, 100$, with a global minimum of 0. For this example, we use a standard BO implementation with a prevalent AF[1], Upper Confidence Bound (UCB) [26], denoted as:

$$\alpha(x) = -\mu(x) + \sqrt{\beta_t} \cdot \sigma(x) \tag{4.1}$$

[1]Other acquisition functions include the probability of improvement (PI) and expected improvement (EI).

where $\mu(x)$ and $\sigma(x)$ are the posterior mean (prediction) and posterior standard deviation (uncertainty) at point x predicted by the surrogate model, and β_t is a hyperparameter that trades off between exploration and exploitation. we set $\beta_t = 1.96$ in this example.

Here, we use random search to create the initial starting points for a *multi-start gradient-based* AF maximiser to iteratively generate multiple candidates, from which the AF chooses a sample for evaluation. In each BO iteration, we first evaluate the AF on 100000 random points and then select the top n points as the initial points for the further gradient-based AF maximiser. We denote this implementation as `BO-grad`. In this example, we use two settings $n = 10$ and $n = 1000$.

As shown in Figure 4.3(a), the function output given by BO-grad with 10 AF maximiser restarts is far from the global minimum of 0. We hypothesise that while the AF is effective, the low quality of candidate samples generated in AF maximisation limits the power of the AF. To verify our hypothesis, we further consider two strategies: (1) either randomly select the next query point from all the candidate points generated during AF maximisation or (2) exhaustively evaluate all candidate points' objective function values at each BO iteration and select the best one as the next query point. The former and latter strategies correspond to "**random selection**" and "**optimal selection**" schemes, respectively. Despite the ideal but costly "optimal selection" search scenario, BO does not converge well, indicating intrinsic deficiencies in the AF maximisation process. Meanwhile, the AF itself can choose a good candidate sample point to evaluate, as the performance of the native AF-based BO-grad is close to that of "optimal selection" and better than that of "random selection". This observation suggests that the AF is effective at selecting a good sample in this case but its power is severely limited by the candidate samples generated during the AF maximisation process. We also observe similar results manifest when using other representative maximisers like random sampling and evolutionary algorithms.

We then test what happens when we increase the number of AF maximisation restarts of BO-grad to generate more candidates for AF to select at each iteration. However, in Figure 4.3(b), it is evident that even with an increase in random restarts to 1000, the quality of intermediate candidate points generated during the AF maximisation process remains similar to that with 10 restarts. Furthermore, in the case of 1,000 restarts, the performance of BO-grad is still close to that of "optimal selection",

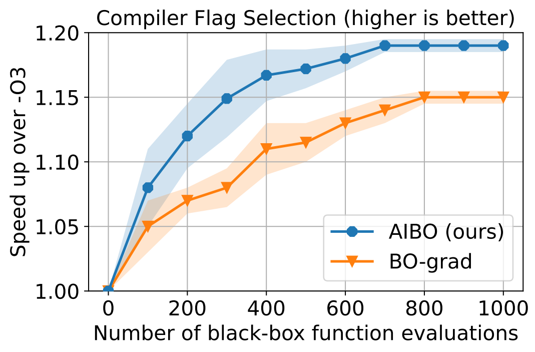


Figure 4.4: Replacing the default initialisation strategy of AF maximisation in BO-grad with the proposed strategy in AIBO leads to improved performance for the compiler flag selection task.

reinforcing our observation that the pool of candidates restricts AF's power. This observation suggests that we need a better initialisation scheme rather than simply increasing the number of restarts of random initialisation.

This example motivates us to explore the possibility of improving the BO performance by providing the AF with better candidate samples through enhanced AF maximiser initialisation. Our intuition is that the potential of AF is often not fully explored in high-dimensional BO. Moreover, the commonly used random initialisation of the AF maximisation process is often responsible for inferior candidates. We aim to improve the quality of the suggested samples through an enhanced mechanism for AF maximiser initialisation. As we will show in Section 4.5, our strategy AIBO significantly improves BO on the 100D Ackley function, finding an output minimum of less than 0.5, compared to 6 given by BO-grad after evaluating 5,000 samples.

4.2.2 Compiler Autotuning

To investigate whether the issue of inadequate AF maximisation exists in the domain of compiler autotuning, we further examine the performance of the proposed strategy in the task of selecting compiler flags to optimise the execution time. Compared to the compiler phase-ordering problem, compiler flag selection is simpler: it does not involve determining the ordering of passes, the interactions between passes are less complex, and each decision dimension is essentially binary (flag on or off). Therefore, Bayesian optimisation can be directly applied to this problem without additional customisation.

For this study, we select the telecom_gsm benchmark from the cBench suite [162] as our test instance, utilising LLVM 17 as the compiler. We chose the telecom_gsm benchmark because it is a representative compute-intensive program, and its compilation and execution times are short enough to allow repeated evaluations required by Bayesian optimisation. The benchmark is run as a single-threaded program on an AMD Ryzen Threadripper PRO 5995WX CPU clocked at 2.25 GHz. We consider 82 flags at the -O3 optimisation level, where each compiler flag can be set to 0 or 1 (0/1 refers to disabling/enabling it). Since the previously mentioned Bayesian optimisation method, BO-grad, is designed for continuous domains, we reformulate the flag selection problem as a continuous optimisation task. Specifically, we embed the binary compiler-flag search space into a continuous domain $[0, 1]^d$ to allow Bayesian optimisation to operate smoothly. Each dimension corresponds to a compiler flag, where values below 0.5 are mapped to 0 (flag disabled) and values above or equal to 0.5 are mapped to 1 (flag enabled). Intermediate values (e.g., 0.45) are not directly meaningful to the compiler; they are only used internally by BO during the optimisation process and are thresholded to obtain the binary flag configuration for evaluation.

We follow the common practice to deal with the noise in program execution time measurements and the relatively long runtime of each execution [67]. At each search iteration, we evaluate the objective function (i.e., program execution time) by running the program three times and taking the average. After the search process concludes, to accurately assess the true performance, we select the best configuration found every 100 points, recompile the program, and execute it 50 times to obtain the accurate execution time.

Figure 4.4 compares BO-grad (10 restarts) and AIBO. Results show that the improved initialisation strategy (of AF maximisation) in AIBO leads to more efficient optimisation than the default strategy in BO-grad. This underscores that when utilising BO for compiler autotuning, the problem of inadequate AF maximisation remains.

4.3 Methodology

Our study focuses on evaluating the initialisation phase of AF maximisation. To this end, we developed AIBO, an open-source framework to facilitate an exhaustive and reproducible assessment of AF maximiser initialisation methods.

4.3.1 Heuristic Acquisition Function Maximizer Initialisation

AIBO leverages multiple heuristic optimisers' candidate generation mechanisms to generate high-quality initial points from the already evaluated samples. Given the proven effectiveness of heuristic algorithms in various black-box optimisation scenarios, they are more likely to create initial candidates near promising regions. As an empirical study, we aim to explore whether this initialisation makes the AF optimiser yield points with higher AF values and superior black-box function values compared to random initialisation.

As described in Algorithm 1, AIBO maintains multiple black-box heuristic optimisers $o_0, o_2, ..., o_{l-1}$ (the size is l). At each BO iteration, each heuristic optimiser o_i is asked to generate k raw points X^i based on its candidate generation mechanisms (e.g., CMA-ES generates candidates from a multivariate normal distribution). AIBO then selects the best n points $\widetilde{X}^i$ ($n < k$) with the highest AF values from X^i for each optimiser o_i, respectively. After using these points to initialise and run an AF maximiser for each initialisation strategy, we obtain multiple candidate points $x_t^0, x_t^1, ..., x_t^{l-1}$. Finally, the candidate with the highest AF value is chosen as the sample to be evaluated by querying the black-box function. Crucially, the evaluated sample is used as feedback to update each optimiser o_i - for example, updating CMA-ES's normal distribution. This process repeats at each subsequent BO iteration.

Our current default implementation employs CMA-ES, GA and random search as heuristics for initialisation. We use the "combine-then-select" approach because it allows us to examine if GA/CMA-ES initialisation could find better AF values than random initialisation. Our scheme only chooses GA/CMA-ES initialisation if it yields larger AF values than random initialisation. Besides, while heuristics like GA already provide exploratory mechanisms and altering their hyperparameters can achieve different trade-offs, the usage of random initialisation here could also mitigate the case of over-exploitation.

CMA-ES CMA-ES uses a multivariate normal distribution $\mathcal{N}(m, C)$ to generate initial candidates in each BO iteration. Here, the mean vector m determines the centre of the sampling region, and the covariance matrix C determines the shape of the region. The covariance matrix m is initialised at the beginning of the BO search, and each direction (dimension) will be assigned an initial covariance, ensuring exploration

Algorithm 1 Acquisition function maximiser initialisation for high-dimensional Bayesian optimisation (AIBO)

```
    Input: The number of search iterations $T$
    Output: The best-performing query point $x^*$
 1: Draw $N$ points uniformly and evaluate their objective values to form the initial dataset $D_0$
 2: Specify a set of heuristic optimisers $\mathcal{O} = \{o_0, o_2, ..., o_{l-1}\}$, where the size is $l$
 3: Use $D_0$ to initialize a set of heuristic optimisers $\mathcal{O}$
 4: for $t = 0 : T - 1$ do
 5:     Fit a Gaussian process $\mathcal{G}$ to the current dataset $D_t$
 6:     Construct an acquisition function $\alpha(x)$ based on $\mathcal{G}$
 7:     for $i = 0 : l - 1$ do
 8:         $\mathbf{X}^i \leftarrow o_i.ask(num = k)$    ▷ Ask the heuristic to generate $k$ candidates
 9:         $\widetilde{\mathbf{X}}^i \leftarrow top(\alpha(\mathbf{X}^i), n)$    ▷ Select top-n $(n < k)$ candidates from $\mathbf{X}^i$ according to $\alpha(x)$
10:         Use $\widetilde{\mathbf{X}}^i$ to initialize an acquisition function maximizer $\mathcal{M}$
11:         $x_t^i \leftarrow \arg\max_{x \in \mathcal{X}} \alpha(x) | \mathcal{M}$    ▷ Use $\mathcal{M}$ to maximize $\alpha(x)$
12:     end for
13:     $x_t \leftarrow \arg\max \alpha(x)\ x \in \{x_t^0, x_t^1, ..., x_t^{l-1}\}$    ▷ Select the point with the highest AF value
14:     $y_t \leftarrow f(x_t)$    ▷ Evaluate the selected sample
15:     for each $o_i \in \mathcal{O}$ do
16:         $o_i.tell(x_t, y_t)$    ▷ Update heuristic optimiser $o_i$ with $(x_t, y_t)$
17:     end for
18:     Update dataset $D_{t+1} = D_t \cup \{(x_t, y_t)\}$
19: end for
```

across all directions. By updating m and C using new samples after each BO iteration, CMA-ES can gradually focus on promising regions.

GA GA keeps a population of samples to determine its search region. It uses biologically inspired operators like mutation and crossover to generate new candidates based on the current population. Its population is updated by newly evaluated samples after each BO iteration.

Random Most BO algorithms or library implementations use random search for initialising the AF maximiser. We use it here to eliminate the possibility of AIBO's performance improvement stemming from GA/CMA-ES initialisation, yielding points with better black-box function values but smaller AF values.

Our heuristic initialisation process is AF-related, as the heuristic optimisers are updated by AF-chosen samples. Usually, a more explorative AF will make the heuristic initialisation also more explorative. For instance, in GA, if the AF formula leans towards exploration, the GA population composed of samples chosen by this AF will have greater diversity, leading to generating more diverse raw candidates. The details of how GA and CMA-ES generate candidates and update themselves are provided in Section 4.3.2.

4.3.2 Implementation Details

Since this study focuses on the AF maximisation process, we utilise other BO settings that have demonstrated good performance in prior work. We describe the implementation details as follows.

Gaussian process regression

To support scalable GP regression, we implement the GP model based on an optimised GP library GPyTorch [22]. GPyTorch implements the GP inference via a modified batched version of the conjugate gradients algorithm, reducing the asymptotic complexity of exact GP inference from $O(n^3)$ to $O(n^2)$. The overhead of running BO with a GP model for a few thousand evaluations should be acceptable for many scenarios that require hundreds of thousands or more evaluation iterations.

We select the Matérn-5/2 kernel with ARD (each input dimension has a separate length scale) and a constant mean function to parameterise our GP model. The details of the Matérn-5/2 kernel have been introduced in the background chapter 2, and we choose it because it is one of the most widely used kernels and has demonstrated strong empirical performance across a broad range of real-world optimisation tasks. The model parameters are fitted by optimising the log-marginal likelihood before proposing a new batch of samples for evaluation. Following the usual GP fitting procedure, we re-scale the input domain to $[0, 1]^d$. To improve model stability, we apply Yeo-Johnson power transforms to function values, which reduces skewness and makes the data more Gaussian-like. This is particularly useful for highly skewed functions such as Rosenbrock and has proven effective in real-world applications [131]. We use the following bounds for the model parameters: length-scale $\lambda_i \in [0.005, 20.0]$, noise variance $\sigma^2 \in [1e^{-6}, 0.01]$, consistent with parameter settings in previous work [7].

Batch Bayesian Optimisation

To support batch evaluation for general high-dimensional problems, we employ the UCB and EI AFs estimated via Monte Carlo (MC) integration. Wilson et al. [28] have shown that MC AFs naturally support queries in parallel and can be maximised via a greedy sequential method. Algorithm 1 shows the case where the batch size is one, which is the typical setting for compiler autotuning tasks. Assuming the batch size is q, the process of greedy sequential acquisition function maximisation can be expressed as follows:

1. Maximize the initial MC acquisition function $\alpha^0(x)$ to obtain the first query point x_0.
2. Use the first query sample $(x_0, \alpha^0(x_0))$ to update $\alpha^0(x)$ to $\alpha^1(x)$ and maximize $\alpha^1(x)$ to obtain the second query point x_1.
3. Similarly, successively update and maximize $\alpha^2(x), \alpha^3(x), ..., \alpha^{q-1}(x)$ and obtain query points $x_2, x_3, ..., x_{q-1}$.

We implemented it based on BoTorch [25], which provided the MC-estimated acquisition functions and the interface for function updating via query samples. Details of the calculation of MC-estimated AFs are provided in Section 2.1.2.

Hyper-parameter settings

We use $N = 50$ samples to obtain all benchmarks' initial dataset D_0. We set $k = 500$ and $n = 1$ for each AF maximiser initialisation strategy. We use the implementations in pycma [163] and pymoo [164] for the CMA-ES and the GA initialisation strategies, respectively. For CMA-ES, we set the initial standard deviation to 0.2. For GA initialisation, we set the population size to 50. The default AF maximiser in AIBO is the gradient-based optimisation implemented in BoTorch. The default AF is UCB with $\beta_t = 1.96$ (default setting in the skopt library [128]), and the default batch size is set to 10. In Section 4.5.7, we will also show the impact of changing these hyper-parameters in our experiments.

Details of GA and CMA-ES in AIBO

In this part, we explain how GA and CMA-ES optimisers in AIBO are initialised (Line 3 in Algorithm 1), updated (Line 16 in Algorithm 1) and asked to generate raw candidates (Line 8 in Algorithm 1). We use the implementations in pycma and pymoo for the CMA-ES and the GA strategies, respectively.

Initialisation of GA At the beginning of the search process in AIBO, we will draw N samples uniformly and evaluate them to produce GA's initial population.

Candidate generation of GA To generate new candidates, GA will sequentially perform selection, crossover and mutation operations. The selection operation aims to select individuals from the current population of GA to participate in mating (crossover and mutation). The crossover operation combines parents into one or several offspring. Finally, the mutation operation generates the final candidates based on the offspring created through the crossover. It helps increase the diversity in the population. The selection, crossover and mutation operations used in AIBO are shown as follows.

- *Tournament Selection*: It involves randomly picking T individuals from the population, comparing their fitness, and selecting the individual with the highest fitness. This process is repeated to fill the new generation. We use the default setting of pymoo, i.e., $T = 2$.
- *Simulated Binary Crossover (SBX)*: This is a widely used crossover technique. A binary notation can represent real values, and then point crossovers can be performed. SBX simulated this operation by using an exponential probability distribution simulating the binary crossover. For this operation, we also use the default SBX implementation of pymoo, where the crossover probability is set to 0.5.
- *Polynomial Mutation*: This mutation follows the same probability distribution as the simulated binary crossover to introduce small, random changes to individuals in the population to maintain genetic diversity. We use the default polynomial mutation implementation of pymoo. The mutation probability is set to 1/D, where D is the dimension of the problem, i.e., the number of design variables.

Update of GA We update the population of GA using the most recently evaluated samples in AIBO. Here, we sort the samples based on their fitness (black-box function value), ultimately retaining those with superior fitness.

Table 4.1: Benchmarks used in evaluation.

	Function/Task	#Dimensions	Search Range
Synthetic	Ackley	20, 100, 300	[-5, 10]
	Rosenbrock	20, 100, 300	[-5, 10]
	Rastrigin	20, 100, 300	[-5.12, 5.12]
	Griewank	20, 100, 300	[-10, 10]
Real-world	Robot pushing [108]	14	/
	Rover trajectory planning [108]	60	[0, 1]
	Half-Cheetah locomotion [165]	102	[-1, 1]
	Nasbench [166]	36	[0, 1]
	Lasso DNA [167]	180	[0, 1]

Initialisation of CMA-ES The key of CMA-ES is a multivariate normal distribution $\mathcal{N}(m_k, \sigma_k^2 C_k)$, which is initialised at the beginning of the search process in AIBO. In particular, we will draw N samples uniformly and evaluate them. The coordinates of the sample with the best black-box function value will be used as the initial mean vector m_0. The step size σ_k is initialized to a constant $\sigma_0 = 0.2$, and the covariance matrix C_k is initialized as an identity matrix $C_0 = I$.

Candidate generation of CMA-ES At each iteration k, CMA-ES generates candidates by sampling from its current multivariate normal distribution, i.e.,

$$x_i \sim \mathcal{N}(m_k, \sigma_k^2 C_k) \tag{4.2}$$

Update of CMA-ES The update of CMA-ES involves updating the multivariate normal distribution $\mathcal{N}(m_k, \sigma_k^2 C_k)$. Here we follow the standard CMA-ES's update process, which is shown in equations 2.8–2.12 in Section 2.2.2.

4.4 Experimental Setup

4.4.1 Benchmarks

Table 4.1 lists the benchmarks and the problem dimensions used in the experiments. These include synthetic functions and six real-world tasks.

Synthetic functions We first apply AIBO and the baselines to four common synthetic functions [161]: Ackley, Rosenbrock, Rastrigin and Griewank[1]. These functions are frequently used in optimisation studies due to their well-understood landscapes and controlled complexity. Each function allows for flexible dimensionality and shares a global minimum value of 0, making them ideal for comparing optimisation algorithms across various settings. The mathematical structures of these functions exhibit different challenges such as multimodality, narrow ridges, and non-convex regions. For instance, the Rastrigin function has a high number of local minima, whereas the Rosenbrock function features a narrow, curved valley. To explore the influence of dimensionality on BO performance, we conduct experiments in 20, 100, and 300 dimensions. This range allows us to assess how AIBO scales with increasing dimensionality while preserving the underlying structure of the problem.

Beyond these synthetic benchmarks, we further evaluate our methods on six real-world tasks to demonstrate their practical applicability.

Robot pushing This task, first introduced in [108] and later considered in [7], aims to optimise the control policy for two robotic hands tasked with pushing two objects to specified target locations. While the dimensionality of this task is relatively low at 14, the sparsity of the reward function poses significant challenges.

Rover trajectory planning The task, also considered in [7, 108], is to maximise the trajectory of a rover over rough terrain. The trajectory is determined by fitting a B-spline to 30 points in a 2D plane (thus, the state vector consists of 60 variables). This task's best reward is 5. Such trajectory optimisation tasks have practical implications in autonomous vehicle navigation and exploration robotics.

Half-cheetah robot locomotion We consider the 102D half-cheetah robot locomotion task simulated in MuJoCo [165] and use the linear policy $a = Ws$ introduced in [168] to control the robot walking. Herein, s is the state vector, a is the action vector, and W is the linear policy to be searched for to *maximise* the reward. Each component of W is continuous and within [-1,1]. This task represents a typical reinforcement learning scenario with continuous control, where the search space is large, and the reward function is often noisy and non-convex.

[1]These functions can be seen in `https://www.sfu.ca/~ssurjano/ackley.html`

Nasbench The Nasbench-101 [166] task is to design a neural architecture cell topology defined by a DAG with 7 nodes and up to 9 edges to maximise the CIFAR-10 test-set accuracy, subject to a constraint where the training time was less than 30 minutes. We follow the parameterisation approach used by [102] to make the Nasbench task become a 36-dimensional black-box function. Note while the dimension is 36, the search space size of Nasbench-101 is small (only 423,624 samples). In such a small search space size, AF maximisation is no longer a challenge.

Lasso-DNA The Lasso-DNA [167] task minimises the mean squared error (MSE) of weighted Lasso sparse regression for real-world DNA datasets. This problem features a high-dimensional search space with 180 tunable parameters, corresponding to the weights of the Lasso regression model. The task is representative of real-world scientific applications where sparse and interpretable models are desired, and optimisation must navigate a highly structured and correlated parameter space. The Lasso-DNA dataset provides a challenging benchmark due to the complex interactions between variables and the need to balance sparsity and predictive accuracy.

Compiler Flag Selection The compiler flag selection task is the same as the example used in Section 4.2.2.

4.4.2 Evaluation Methodology

We design diverse experiments to validate the significance of the initialisation of the AF maximisation process. All experiments are run 50 times for evaluation. In Section 4.5.1, we evaluate AIBO's end-to-end BO performance by comparing it to various baselines including standard BO implementations with AF maximiser random initialisation, heuristic algorithms and representative high-dimensional BO methods. In Section 4.5.2, we evaluate the robustness of AIBO under different AFs. In Section 4.5.3, we evaluate AIBO's three initialisation strategies in terms of AF values, GP posterior mean, and GP posterior variance to show how pure random initialisation leads to over-exploration. This will also show whether AIBO's heuristic initialisation strategies lead to better AF maximisation compared to commonly used random initialisation. In Section 4.5.4, we use an extreme case to test whether adding random initialisation to AIBO can help alleviate the over-exploitation issue. In Section 4.5.5, we use ablation experiments to examine the impact of the three individual initialisation strategies in

AIBO. In Section 4.5.6, we compare AIBO to BO implementations with alternative AF maximiser initialisation strategies rather than selecting initial points with the highest AF values from a set of random points. In Section 4.5.7, we show the impact of hyper-parameters on the performance of AIBO. In Section 4.5.8, we discuss how different AF settings affect the evolution of AIBO's heuristic initialisation strategies, providing evidence that our initialisation process is indeed AF-dependent. Given that different AF settings represent varying trade-offs between exploration and exploitation, we argue that an appropriate initialisation strategy should be adjusted based on the chosen acquisition function, and our approach satisfies this requirement. In Section 4.5.9, we provide the algorithmic runtime of our method.

4.5 Experimental Results

Highlights of our evaluation results are:

- AIBO significantly improves standard BO and outperforms heuristic algorithms and representative high-dimensional BO methods in most test cases (Sec. 4.5.1 and Sec. 4.5.2);
- By investigating AIBO's three initialisation strategies in terms of AF maximisation, we show that random initialisation limits AFs' power by yielding lower AF values and larger posterior variances, leading to over-exploration, empirically confirming our hypothesis (Sec. 4.5.3);
- We provide a detailed ablation study and hyper-parameters analysis to understand the working mechanisms of AIBO (Sec. 4.5.5 and Sec. 4.5.7).
- We demonstrate the importance of leveraging the black-box optimisation history in the AF maximisation process (Sec. 4.5.6).

4.5.1 Comparison with Baselines

Setup

We first compare AIBO to eight baselines: BO-grad, BO-es, BO-random, TuRBO, HeSBO, CMA-ES, GA and AIBO-none. We describe the baselines as follows.

BO-grad, BO-es and BO-random respectively refer to the standard BO implementations using three AF maximisers: multi-start gradient-based, CMA-ES and random

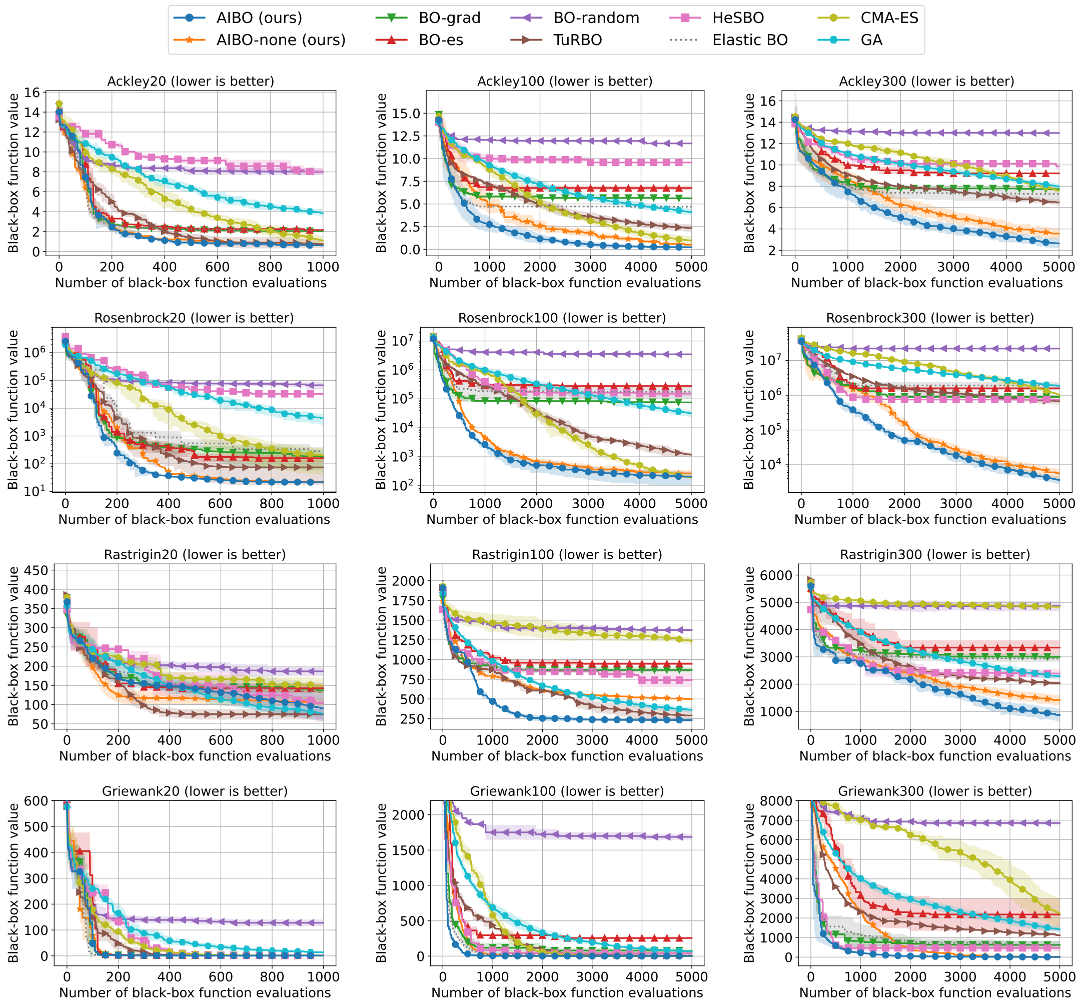


Figure 4.5: Results on synthetic functions. AIBO consistently improves BO-grad on all test cases and outperforms other competing baselines in most cases. Especially, AIBO shows clear advantages in higher dimensions (100D and 300D). Note that AIBO differs from BO-grad only in the initialisation of the AF maximiser.

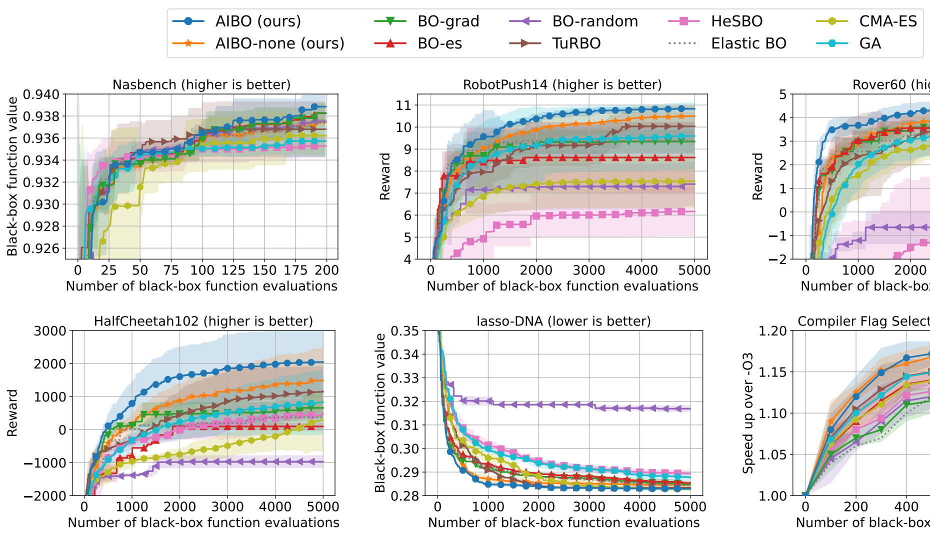


Figure 4.6: Results on real-world problems. AIBO consistently improves BO-grad on all test cases and outperforms other competing baselines in most cases.

sampling. These standard BO implementations use the same base settings as AIBO but with a random initialisation scheme for AF maximisation. To show the effectiveness of AIBO, BO-grad is allowed to perform more costly AF maximisation; we set $k = 2000$ and $n = 10$.

TuRBO [7], HeSBO [100] and Elastic BO [113] are representative high-dimensional BO methods. We use $N = 50$ samples to obtain the initial dataset D_0 for all three high-dimensional BO methods. For HeSBO, we use a target dimension of 8 for the 14D robot pushing task, 20 for the 102D robot locomotion task and 100D or 300D synthetic functions, and 10 for other tasks. Other settings are default in the reference implementations.

CMA-ES and GA are used to demonstrate the effectiveness of AF itself. Given that AIBO employs AF to further search the query point from the initial candidates generated by CMA-ES and GA black-box optimisers, if the AF is not sufficiently robust, the performance of AIBO might be inferior to CMA-ES/GA. For CMA-ES, the initial standard deviation is set to 0.2, and the rest of the parameters are defaulted in pycma [163]. For GA, the population size is set to 50, and the rest of the parameters are defaulted in pymoo [164].

AIBO-none is a variant of AIBO. In each BO iteration, following the initialisation of the AF maximisation process, AIBO-none directly selects the point with the highest AF value while AIBO uses a gradient-based AF maximiser to further search points with higher AF values. This comparison aims to assess whether better AF maximisation can improve performance.

Results

Figures 4.5 and 4.6 present a comprehensive comparison of AIBO against various baselines on synthetic functions and real-world benchmarks, respectively. We use UCB1.96 (UCB with $\beta_t = 1.96$) as the default AF. The results are averaged over 50 runs. The y-axis shows the best function value found so far, and the x-axis shows the number of function evaluations. The shaded area represents the standard deviation.

AIBO versus BO-grad While the performance varies across target functions, AIBO consistently improves BO-grad on all test cases. Especially for synthetic functions which allow flexible dimensions, AIBO shows clear advantages in higher dimensions (100D and 300D). We also observe that BO-grad exhibits a similar convergence rate to AIBO at the early search stage. This is because AF maximisation is relatively easy to fulfil when the number of samples is small. However, as the search progresses, more samples can bring more local optimums to the AF, making the AF maximisation process increasingly harder.

AIBO versus CMA-ES/GA As AIBO introduces CMA-ES and GA black-box optimisers to provide initial points for AF maximisation, comparing AIBO with CMA-ES and GA will show whether the AF is good enough to make the AF maximisation process find better points than the initial points provided by CMA-ES/GA initialisation. Results show AIBO outperforms CMA-ES and GA in most cases except for the 20D Rastrigin function, where GA shows superior performance. However, in the next section, we will demonstrate that adjusting UCB's beta from 1.96 to 1 will enable AIBO to maintain its performance advantage over GA. This suggests that with the appropriate choice of the AF, BO's model-based AF can offer a better mechanism for trading off exploration and exploitation compared to heuristic GA/CMA-ES algorithms.

AIBO versus other high-dimensional BO methods When compared to representative high-dimensional BO methods, including TuRBO, Elastic BO and HeSBO, AIBO performs the best in most cases except for the 20D Rastrigin function, for which TuRBO shows the fastest convergence. However, for higher dimensions (100D and 300D), AIBO performs better than TuRBO on this function.

AIBO versus AIBO-none Without the gradient-based AF optimiser, AIBO-none still shows worse performance than AIBO. This indicates that better AF maximisation can improve the BO performance. This trend can also be observed in the results of standard BO with different AF maximisers, where BO-grad and BO-es outperform BO-random.

Overall, these experimental results highlight the importance of the AF maximisation process for high-dimensional BO, as simply changing the initialisation of the AF maximisation process brings significant improvement.

4.5.2 Evaluation under Different AFs

We also evaluate the performance of AIBO and BO-grad under different AFs. Besides the default AF setting UCB1.96 (UCB with $\beta_t = 1.96$), we also select UCB1 ($\beta_t = 1$), UCB4 ($\beta_t = 4$) and EI as the AF, respectively. This aims to provide insights into how well AIBO enhances BO-grad across different AF settings, shedding light on its robustness and effectiveness across diverse contexts.

Figure 4.7 shows a comprehensive evaluation of the effectiveness of our AIBO method across various AFs. Changing the AF has a noticeable impact on performance, highlighting the importance of AF selection. If an inappropriate AF is used, such as using UCB4 in Rastrigin20, the performance improvements achieved through the use of AIBO remain highly limited. Despite that, the results we obtained are highly encouraging. While different AFs exhibit varying convergence rates, we consistently observe a noteworthy enhancement in the performance of our method when compared to the standard BO-grad approach. The advantage is clearer in higher dimensions (100D and 300D) than in lower dimensions (20D). These findings highlight the robustness and effectiveness of our initialisation method across different AFs.

4.5.3 Over-Exploration of Random Initialisation

The aforementioned experimental results have demonstrated that heuristic AF maximiser initialisation in AIBO leads to significant end-to-end BO performance improve-

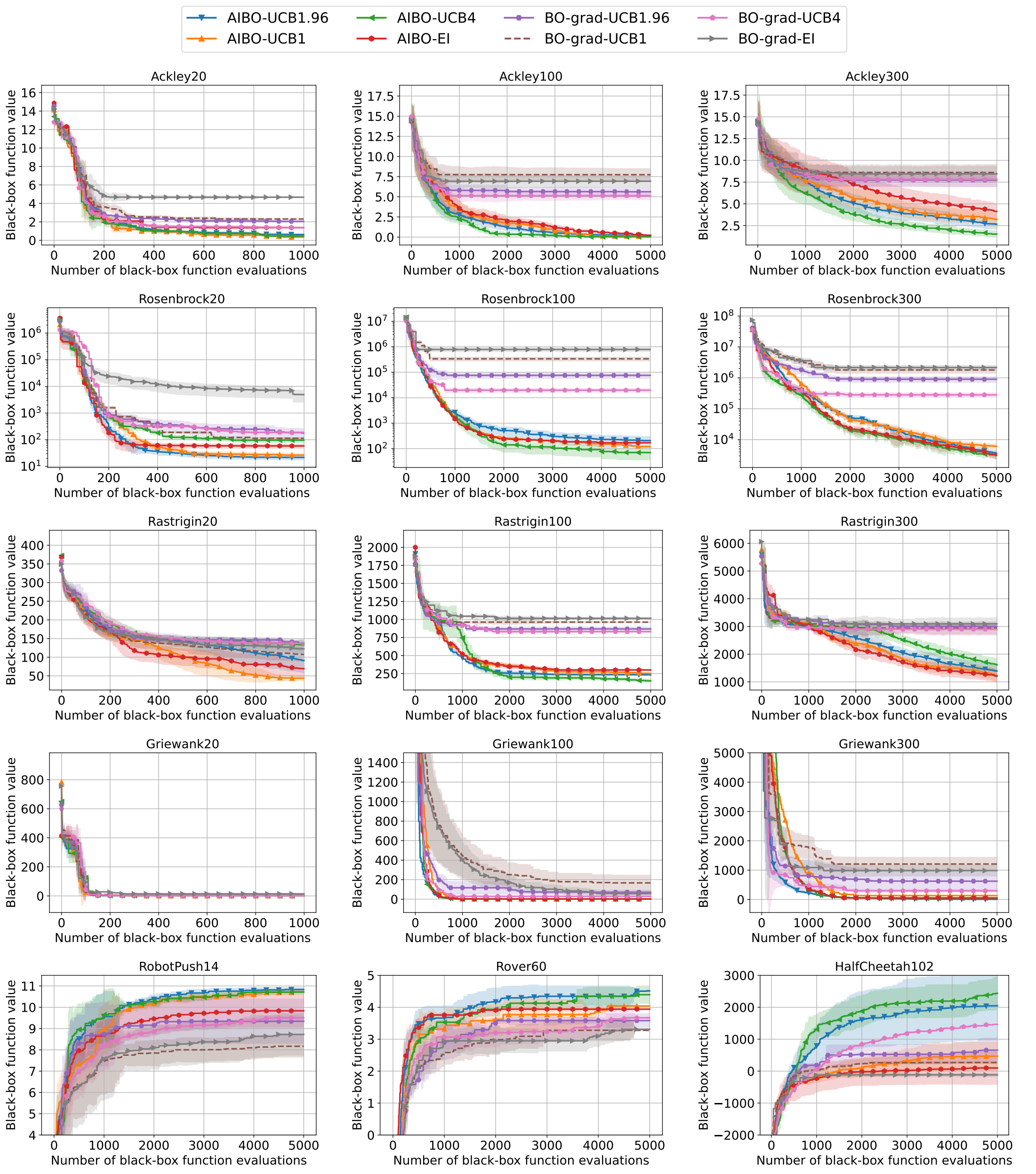


Figure 4.7: Evaluating the performance of AIBO and BO-grad under different AFs on both synthetic functions (lower is better) and real-world problems (higher is better).

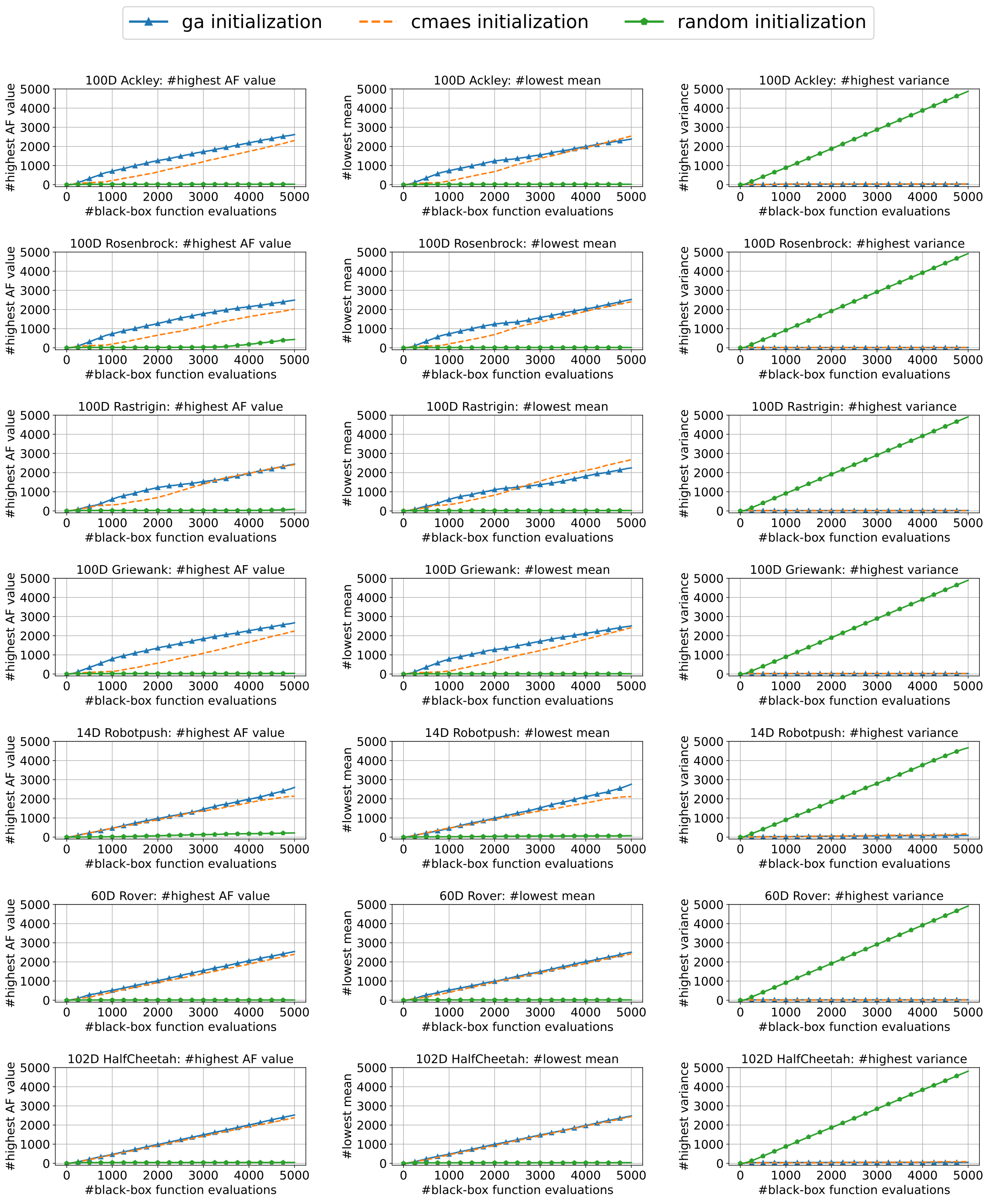


Figure 4.8: Evaluating AIBO's three initialisation strategies in terms of AF values, GP posterior mean, and GP posterior variance when using UCB1.96 as the AF. The **left column** shows the number of times each initialisation achieves the highest AF value among all the three strategies throughout the search process. Similarly, the **middle column** and **right column** indicate instances of achieving the lowest posterior mean (exploitation) and the highest posterior variance (exploration), respectively.

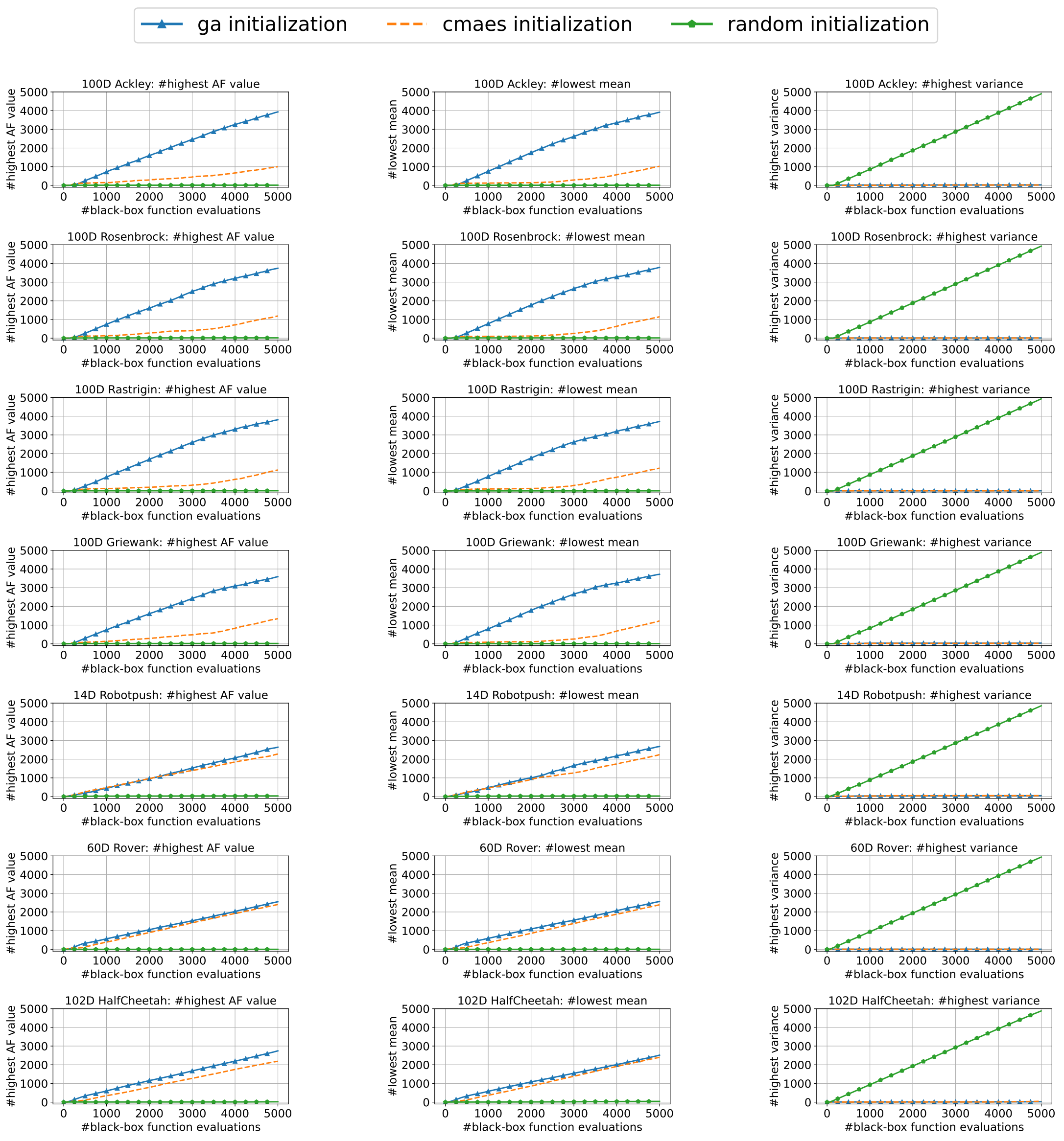


Figure 4.9: Evaluating AIBO's three initialisation strategies in terms of AF values, GP posterior mean, and GP posterior variance when using UCB1 as the AF. The **left column** shows the number of times each initialisation achieves the highest AF value among all the three strategies throughout the search process. Similarly, the **middle column** and **right column** indicate instances of achieving the lowest posterior mean (exploitation) and the highest posterior variance (exploration), respectively.

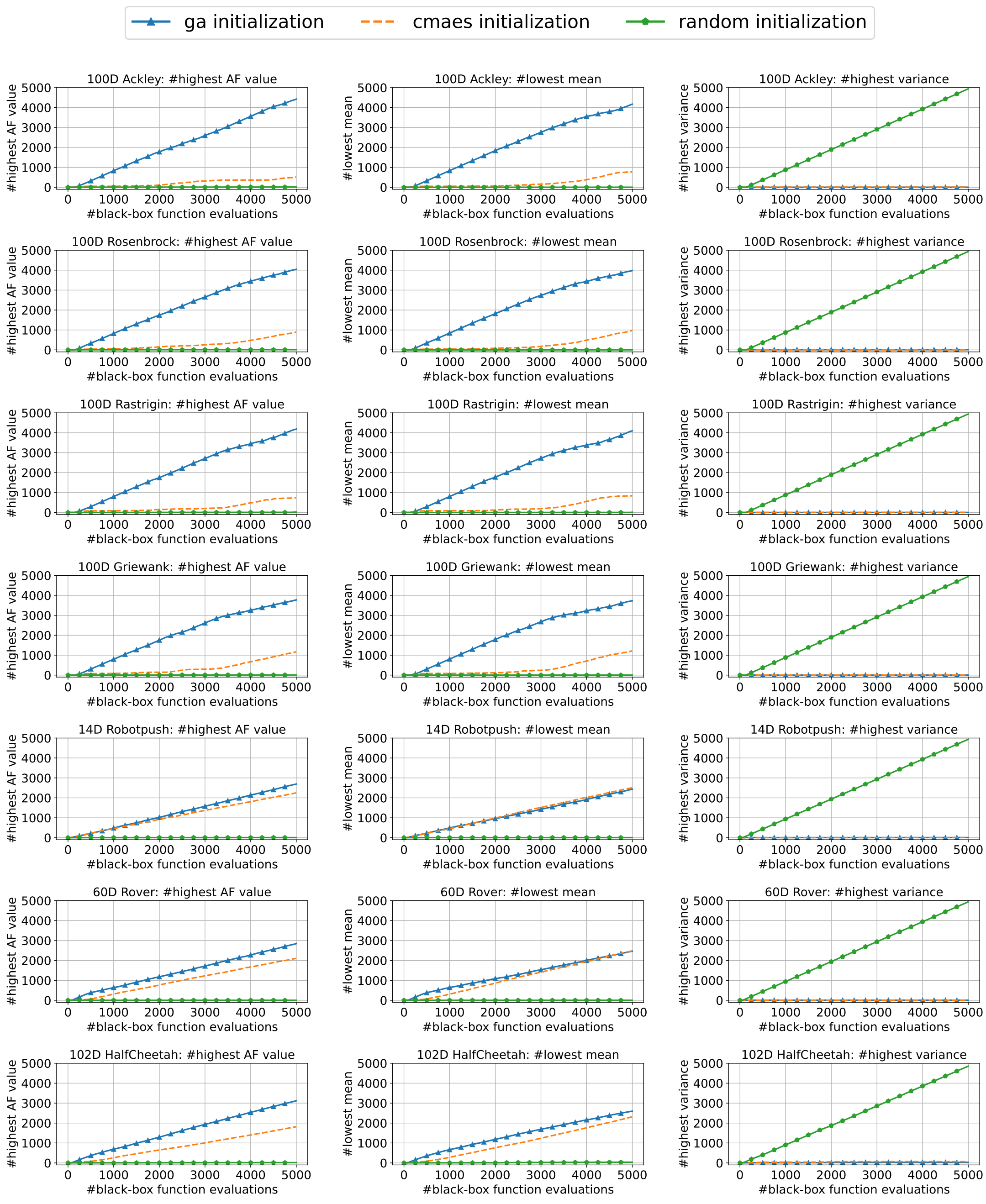


Figure 4.10: Evaluating AIBO's three initialisation strategies in terms of AF values, GP posterior mean, and GP posterior variance when using EI as the AF. The **left column** shows the number of times each initialisation achieves the highest AF value among all the three strategies throughout the search process. Similarly, the **middle column** and **right column** indicate instances of achieving the lowest posterior mean (exploitation) and the highest posterior variance (exploration), respectively.

ments compared to random initialisation. In this subsection, we evaluate AIBO's three initialisation strategies in terms of AF values, GP posterior mean, and GP posterior variance under different AF settings.

In each iteration of AIBO, each initialisation o^i yields a candidate x_t^i after AF maximisation (Line 11 in Algorithm 1). For each initialisation, we count the number of times x_t^i achieves the highest AF value among $\{x_t^0, x_t^1, x_t^2\}$ until the current iteration. This number will show what initialisation dominates the search process by yielding the highest AF value. Similarly, we also count the number of times x_t^i achieves the lowest GP posterior mean (exploitation) and highest GP posterior variance (exploration), respectively. This will examine how different initialisation schemes trade-off between exploration and exploitation.

The left column in Figure 4.8 shows the number of times each initialisation achieves the highest AF value among all the three strategies throughout the search process when using UCB1.96 ($\beta_t = 1.96$) as the AF. The middle and right columns indicate the number of times each initialisation achieves the lowest posterior mean (exploitation) and the highest posterior variance (exploration), respectively. Compared to CMA-ES/GA initialisation, random initialisation always yields lower AF values and higher posterior variance, leading to over-exploration.

This over-exploration caused by random initialisation is not exclusive to the UCB1.96 AF. As shown in Figures 4.9 and 4.10, when decreasing β_t from 1.96 to 1, or using EI as the AF, random initialisation still yields lower AF values and higher posterior variance. This is due to the curse of the dimensionality. Since the search space size grows much faster than sampling budgets as the dimensionality increases, most regions are likely to have a high posterior variance. Given that more samples can bring more local optimums to AFs as the search progresses, random initialisation tends to guide the AF maximiser to find local optimums in regions of high posterior variance. Even if the AF is designed to prioritise regions with lower GP posterior mean values for exploitation (e.g. UCB with a lower β_t), these regions are sparse and may be inaccessible through random initialisation. AIBO is designed to mitigate the drawback of random initialisation, and the results presented here validate AIBO indeed achieves better AF maximisation by optimising the initialisation phase.

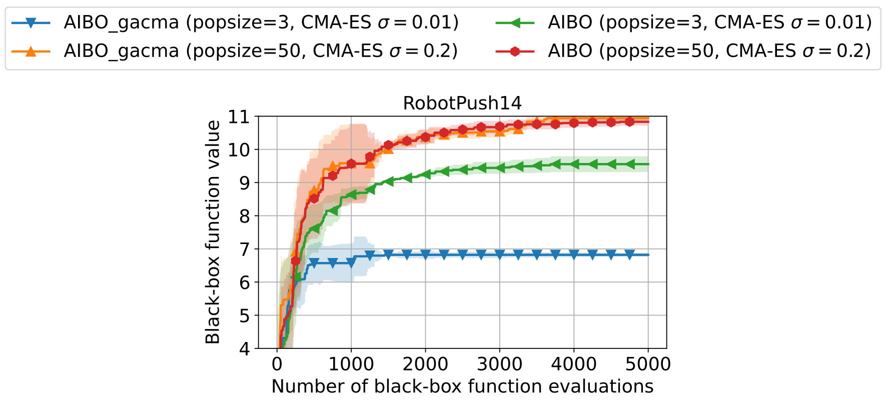


Figure 4.11: Comparision of AIBO and AIBO_gacma under a standard hyperparameter setting and a over-exploitative setting. Results show that random initialisation could help mitigate over-exploitation.

4.5.4 The Case of Over-Exploitation

To show AIBO's random initialisation can help alleviate the over-exploitation issue, we create a variant of our technique, AIBO_gacma, by removing random initialisation. As shown in Figure 4.11, using default hyperparameters, AIBO_gacma performs well in the RobotPush14 optimisation task. However, after adjusting the hyperparameters to an over-exploitation case by setting GA population size to 3 and CMA-ES initial standard deviation to 0.01, we observe that AIBO_gacma performs less effectively. Upon reintroducing random initialisation, we observed significant performance improvement, suggesting that random initialisation could help mitigate over-exploitation.

4.5.5 Ablation Study

To better understand the role played by each initialisation strategy in AIBO, we evaluate the three individual initialisation strategies in AIBO, leading to three variants of AIBO: AIBO_ga, AIBO_cmaes and AIBO_random. We note that AIBO_random is equivalent to BO-grad discussed earlier. Our fourth variant, AIBO_gacma, removes the random initialisation strategy in AIBO.

As shown in Figure 4.12, advanced heuristic initialisation strategies like GA and CMA-ES show better performance than random initialisation in most cases, showing the advantage of a heuristic algorithm over random initialisation. Using a single ad-

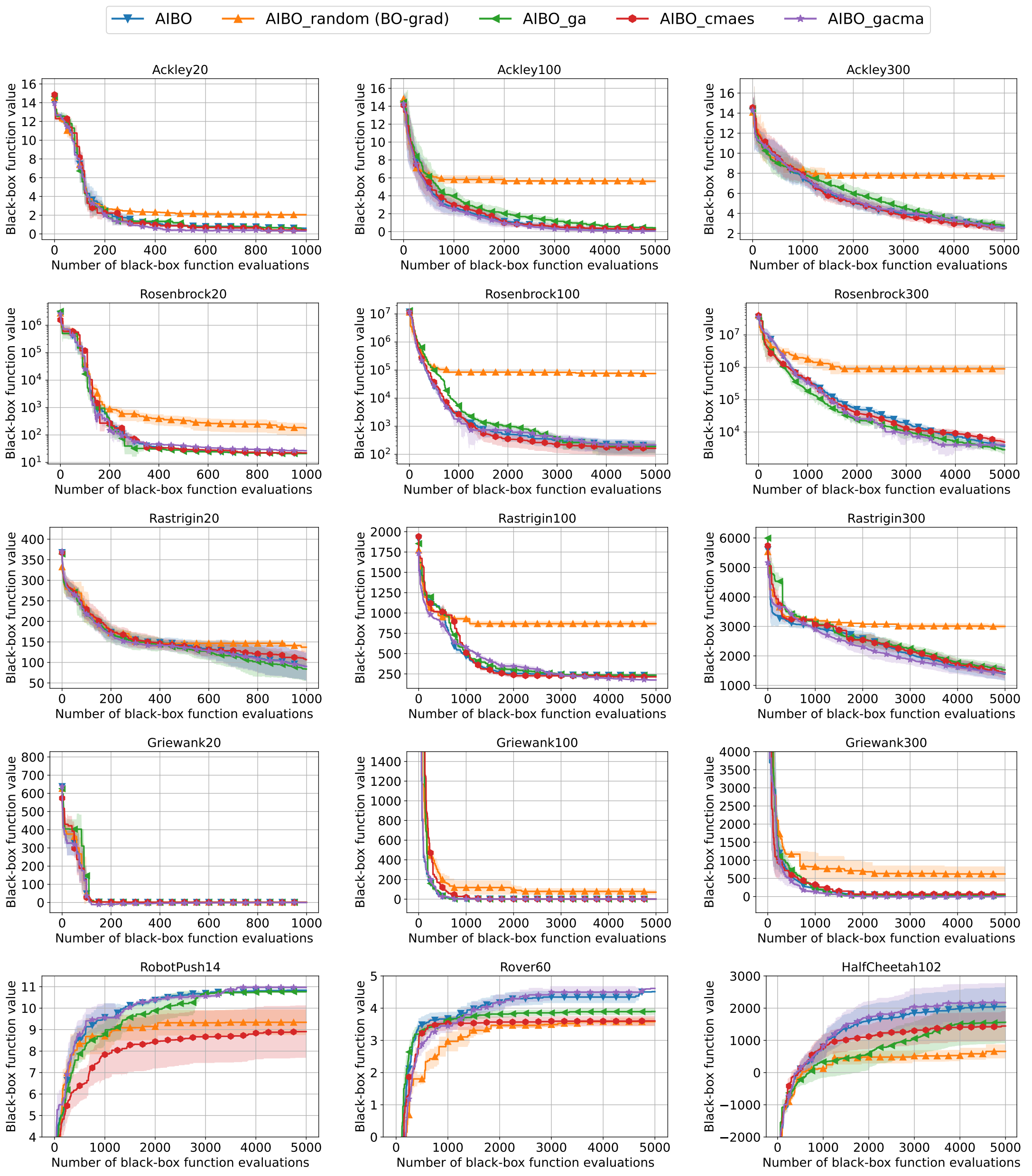


Figure 4.12: Comparing AIBO to its variants AIBO_gacma, AIBO_ga, AIBO_cmaes and AIBO_random (BO-grad) . While a single advanced heuristic heuristic strategy CMA-ES/GA already performs well in most cases, using the ensemble strategy improves the robustness.

vanced heuristic initialisation, AIBO_ga and AIBO_cmaes achieve similar performance to AIBO in most cases. This suggests that CMA-ES and GA can be the main source of performance improvement for AIBO. AIBO_gacma shows a similar performance to AIBO in all cases. This is because GA/CMA-ES initialisation dominates AIBO's search process.

Besides, although AIBO_cmaes is competitive in most problems, it is ineffective for the 14D robot pushing problem, suggesting there is no "one-fits-for-all" heuristic across tasks. By incorporating multiple heuristics, the ensemble strategy used by AIBO gives a more robust performance than the individual components.

4.5.6 Comparison with Other Initialisation Strategies

In previous experiments, we implemented random initialisation by selecting the top-n points with the highest AF values as the initial points from a large set of random points. Some existing BO works have implemented random initialisation in other ways [1] or employed non-random initialisation strategies. The impact of these methods has not been systematically evaluated. We conduct a comparison of the following methods alongside AIBO: BO-cmaes_grad, BO-boltzmann_grad and BO-Gaussian_grad.

Setup

BO-cmaes_grad uses CMA-ES to optimise the AF to provide better initial points for further gradient-based AF maximisation. We note that in this case, CMA-ES is used directly for AF optimisation. In contrast, the "CMA-ES" in AIBO is used to provide initial points by leveraging the history of black-box optimisation. Comparing these two methods will reveal the importance of the black-box optimisation history in the AF maximisation process.

BO-boltzmann_grad refers to the default implementation in BoTorch [25], which uses Boltzmann sampling to generate initial points for the gradient-based AF maximisation. In each BO iteration, it evaluates the AF on a large set of random points and then uses an annealing heuristic (rather than top-n) to select the restart points.

BO-Gaussian_grad uses a Gaussian spray around the incumbent best to generate initial points for the gradient-based AF maximiser. This initialisation strategy has

[1] For example, a popular BO package BoTorch implements random initialisation by using an annealing scheme rather than the top-n to select initial points from a large set of random points.

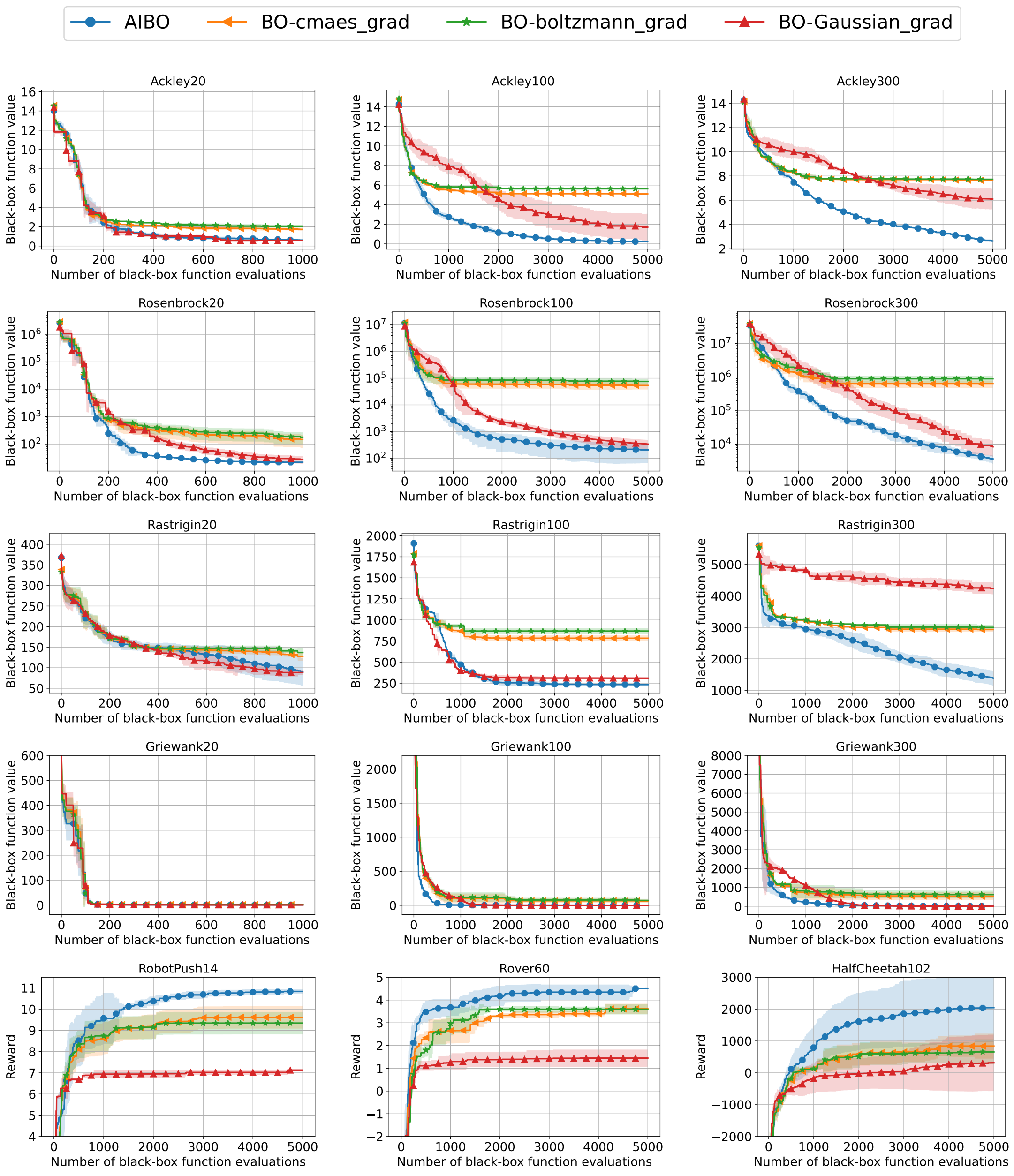


Figure 4.13: Comparing AIBO to standard BO with other AF initialisation methods that do not use random search on both synthetic functions (lower is better) and real-world problems (higher is better).

been used in Spearmint [19], and we replace the AF maximiser with the advanced gradient-based method for a fair comparison.

Results

Figure 4.13 presents the comparison result between AIBO and other initialisation strategies. BO-cmaes_grad and BO-boltzmann_grad exhibit significantly inferior performance compared to AIBO. Both approaches do not leverage prior black-box optimisation history and instead attempt to optimise the AF in the global space directly to provide initial points for further gradient-based AF optimisation. This underscores the challenges of AF optimisation in high-dimensional problems and the importance of utilising the black-box optimisation history. BO-Gaussian_grad takes into account the best points from the past black-box optimisation history as a basis for maximising the AF. This approach performs well in some cases (e.g., Rastrigin100) but may lead to a significant performance drop in other situations (e.g., Robotpush14) due to over-exploitation. Overall, AIBO exhibits significantly better performance compared to these non-random initialisation strategies.

4.5.7 Evaluation under Different Hyper-Parameters

Multiple hyper-parameters in AIBO, including GA population size, CMA-ES initial standard deviation σ, the number of raw candidates generated from heuristics k, the number of selected initial points n, and the batch size could impact its performance.

GA pop size and CMA-ES σ As AIBO employ heuristics to initialise the AF maximisation process, these heuristics' hyper-parameters control the quality of initial points of the AF maximisation process and affect the trade-off between exploration and exploitation. A larger GA population size and a larger CMA-ES initial standard deviation will encourage more exploration. As shown in the left column of Figure 4.14, different tasks favour different trade-offs. A more exploratory setting (popsize=100 and $\sigma = 0.5$) works well for the HalfCheetah102 task but reduces the performance on Ackley100.

k and n Based on Algorithm 1, increasing the number of raw candidates generated from heuristics k and the number of selected initial points n might help AF maxim-

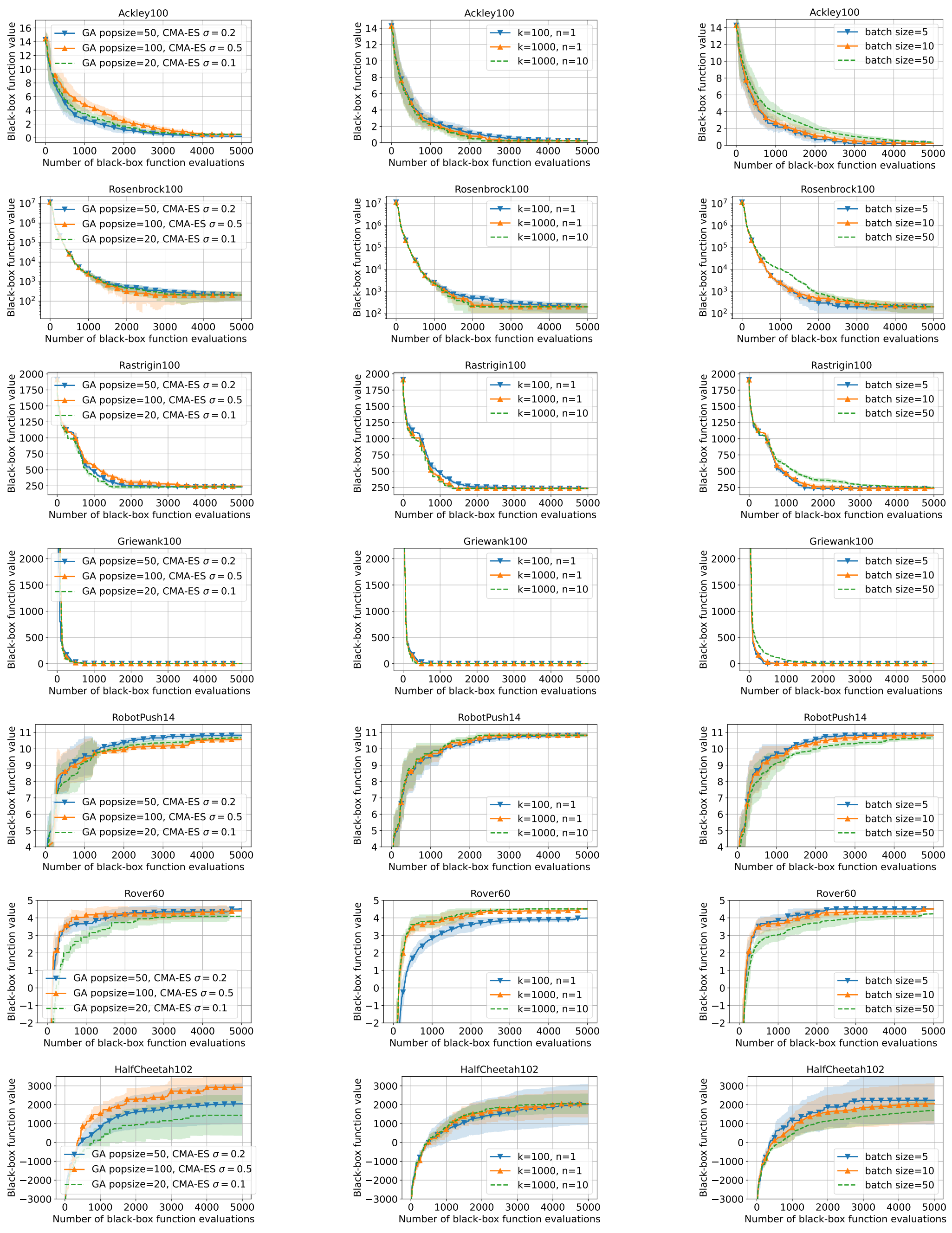


Figure 4.14: The impact of hyper-parameters on AIBO. The **left column** shows the impact of GA population size and CMA-ES initial standard deviation σ. The **middle column** reports the impact of the number of raw candidates generated from heuristics k and the number of selected initial points n. The **right column** shows the impact of the batch size.

ization but requires more calculation. However, as shown in the middle column of Figure 4.14, increasing k and n does not yield significant performance improvement in most cases except for the Rover60 task.

Batch size As shown in the right column of Figure 4.14, AIBO performs well across different batch sizes, and reducing the batch size can slightly enhance convergence speed in all cases.

4.5.8 Impact of AF Settings on GA Population Diversity

Our heuristic initialisation process is AF-related, as it depends on past samples selected by the given AF. Usually, a more explorative AF setting will also make the heuristic initialisation in AIBO more explorative. For instance, if the AF formula leans towards exploration in GA, the GA population composed of samples chosen by this AF will have greater diversity, generating more diverse raw candidates.

To illustrate this point, we select two different AF settings: UCB1.96 ($\beta_t = 1.96$), and UCB9 ($\beta_t = 9$, more exploratory). When applying AIBO to Ackley100, we calculate the average distance between individuals in the GA population at each BO iteration, which is a good measure of the population diversity. We repeated this experiment 50 times. As shown in Figure 4.15, AIBO with UCB9 achieves more diverse GA populations than AIBO with UCB 1.96, suggesting that a more explorative AF setting will make GA initialisation more explorative.

4.5.9 Algorithmic Runtime

In Table 4.2, we provide the algorithmic running time (excluding the time spent evaluating the objective function) for our method with a batch size of 10. For comparison, we also show the algorithmic runtime of BO-grad. The experiments are run on an NVIDIA RTX 3090 GPU equipped with a 20-core Intel Xeon Gold 5218R CPU Processor. As described in Sec 4.5.1, to show the effectiveness of AIBO, BO-grad is allowed to perform more costly AF maximisation. AIBO uses less algorithmic runtime because it costs less AF maximisation time than the standard BO-grad method. AIBO's algorithmic runtime is also acceptable for actual expensive black-box optimisation tasks (only several hours for a few thousand evaluations).

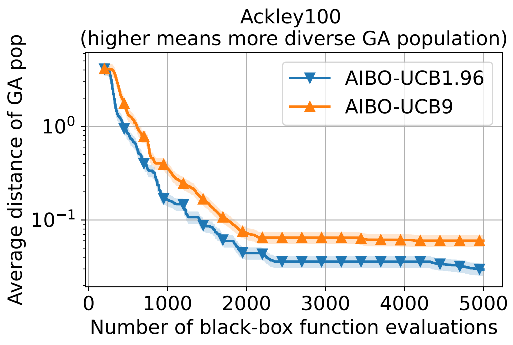


Figure 4.15: Evaluating the average distance between individuals in the GA population when applying AIBO-UCB1.96 and AIBO-UCB9 (more exploratory) to the Ackley100 function. With the same number of BO iterations, AIBO-UCB9 always owns a more diverse GA population than AIBO-UCB1.96, suggesting that a more exploratory AF setting will also make the GA initialisation in AIBO more exploratory.

Table 4.2: Algorithmic runtime

	Synthetic			RobotPush	Rover	HalfCheetah
Dimensions	20	100	300	14	60	102
#Samples	1000	5000	5000	5000	5000	5000
AIBO	8 min	2.5 h	3.6 h	1.8 h	2 h	2.5 h
BO-grad	12 min	3.3 h	5 h	2.5 h	3 h	4 h

4.6 Summary

This chapter aims to understand the challenges of high-dimensional BO. It presents a large-scale empirical study investigating the impact of the acquisition function (AF) maximiser initialisation process in Bayesian optimisation (BO) for high-dimensional problems. Our extensive experiments reveal that the choice of AF maximiser initialisation significantly influences the AF's realisation, and the commonly used random initialisation strategy may fail to fully exploit an AF's potential.

To address this, we introduce AIBO, a framework designed to optimise the initialisation phase of AF maximisation in BO. AIBO is specifically developed to overcome the limitations of random initialisation in high-dimensional BO. It adopts a simple yet effective approach by leveraging multiple heuristic optimisers to generate raw samples for the AF maximiser, thereby achieving a better balance between exploration and exploitation.

We evaluate AIBO on synthetic benchmark functions as well as real-world tasks, including a compiler flag selection problem. Experimental results demonstrate that AIBO significantly enhances the performance of standard BO in high-dimensional settings and outperforms existing high-dimensional BO methods.

Theoretical studies on BO convergence have often relied on the key assumption that the AF can be globally optimised. While this assumption generally holds in low-dimensional scenarios, our large-scale empirical analysis suggests that it may not be valid in high-dimensional problems. We show that the way the AF is optimised plays a crucial role in determining the overall effectiveness of the BO method.

Overall, it proposes a general framework for high-dimensional BO that can be applied to a wide range of tasks including compiler autotuning, thus addressing the high-dimensional challenge (Section 1.2.1) in compiler phase ordering. The following chapter will consider the real application of AIBO in the context of compiler phase ordering optimisation to address other challenges caused by the unique characteristics of the compiler phase ordering task.

Chapter 5

Leveraging Compilation Statistics for Compiler Phase Ordering via Bayesian Optimisation

The chapter presents a customised Bayesian optimisation (BO) method, Citroen, to accelerate compiler phase ordering. Based on the discrete adaptation of the high-dimensional BO method proposed in Chapter 4, Citroen further leverages pass-related compilation statistics to model the complex interactions between compiler passes and uses a dynamic strategy to allocate the search budget across multiple source files within a single program. The rest of the chapter is organised as follows: Section 5.2 provides a detailed motivation example, Section 5.3 presents the proposed approach, Sections 5.4 and 5.5 describe the experimental setup and results, and Section 5.6 concludes the chapter.

5.1 Introduction

A significant challenge in phase ordering is the vast optimisation space. For example, LLVM 17 offers over 100 transformation passes, leading to an extremely large number of possible ways for applying these passes - combinations that would take many machine years to explore exhaustively. Although certain sequences might significantly outperform compiler default settings, these pass sequences can be sparse [4], making them hard to find in such a large space.

Search-based autotuning is widely used for phase ordering [3, 43, 44, 49, 54, 56, 58, 62, 67, 71, 169]. Unlike predictive modelling [81, 83, 87, 89], which can only be applied to a limited number of compiler passes or parameters due to the difficulty in collecting sufficient training samples, search-based methods can be applied to arbitrary compiler pass sequences. However, while this flexibility can be advantageous, finding the optimal compiler pass sequence through search can be prohibitively expensive.

Our work aims to improve the efficiency of search-based autotuning methods for phase ordering. A key drawback of existing search-based approaches for compiler phase ordering is their difficulty in capturing the complex interactions between compiler passes and the order in which they are applied. Identifying which passes positively impact performance during the search allows the algorithm to focus on compiler pass sequences that are more likely to be beneficial. Unfortunately, modelling the effect of a compiler pass is challenging, as its impact depends not only on the input program but also on the other passes it interacts with and the order in which they are applied. For instance, loop unrolling can influence the effectiveness of register allocation and instruction scheduling.

Furthermore, when optimising programs with multiple source files (referred to as *modules* in this work), we aim to apply module-specific pass sequences rather than relying on a 'one-size-fits-all' pass setting for all files. Achieving this requires an adaptive, dynamic strategy for allocating the search budget (i.e., the number of runtime measurements in this work) across different modules, ensuring the search time is used efficiently to maximise the overall performance gains within the available budget.

We present Citroen, a better search-based autotuning method for compiler phase ordering. Our key insight is that pass-related compilation statistics, collected during the execution of compiler passes, can offer valuable information to model pass interactions and guide the search process. For instance, LLVM's *loop-vectorise* pass reports how many loops have been vectorised. If a strong positive correlation is observed between the number of vectorised loops and improved performance from the historical evaluation data collected during the online search, we can infer that loop vectorisation is likely to benefit the input program. In such cases, if changing the compiler pass sequences leads to a reduction in the number of vectorised loops, it suggests that this pass sequence may negatively affect performance. This avoids profiling the binary generated by this pass sequence, thus saving search time.

Citroen is designed to leverage compilation statistics provided by modern compiler infrastructures to accelerate phase ordering autotuning by avoiding the profiling of pass sequences that offer no performance gain. This is achieved through a customised

Bayesian optimisation (BO) method [155], which builds an online probabilistic cost model (known as *the surrogate model*) to evaluate candidate pass sequences. These candidate pass sequences are generated by applying the discrete variant of the heuristic acquisition function (AF) maximiser initialisation strategy introduced in Chapter 4. Our cost model takes as input a feature vector consisting of compilation statistics and predicts both the execution time and the prediction uncertainty. The cost model is dynamically and constantly updated during the search process using new observations so that it becomes more accurate as the search progresses.

CITROEN leverages an acquisition function to avoid profiling sequences that are likely to result in poor performance, while encouraging exploration in regions where the model's predictions are uncertain. For multi-module programs, CITROEN trains the cost model globally by concatenating the compilation statistics of individual source files. This enables the system to dynamically determine which module holds the most potential for performance gains and to allocate the search budget accordingly.

A key distinction between CITROEN and previous BO approaches in compiler optimisation [61, 140, 141, 143] lies in the way the cost model is constructed. In prior works, the standard BO process is employed, using raw tuning parameters as inputs to fit the cost model. These parameters, such as the number of OpenMP threads [141], enabling or disabling a compiler flag [61], loop tile sizes [140], and loop unroll factors [143], have a direct and often predictable impact on performance. However, in the compiler phase-ordering problem, interactions between passes introduce a significantly higher level of complexity, making it much more challenging to anticipate performance gains based on the sequence of passes. CITROEN mitigates the issue using compilation statistics as a proxy to capture the compiler pass interactions. Moreover, the search spaces in prior BO studies on compiler optimisation are significantly smaller than that of phase ordering, removing the necessity to design specialised high-dimensional BO approaches. Instead, CITROEN extends the high-dimensional BO method proposed in Chapter 4 to a discrete version to handle the high-dimensional challenge.

We evaluate CITROEN by applying it to optimise the phase ordering of the LLVM compiler. To ensure the correctness of the program after phase ordering, we apply differential testing, a technique that compares the outputs of the original and optimised programs on the same inputs to detect semantic deviations. We test the resulting compilation system on the cBench [162] and SPEC CPU 2017 [16] benchmark suites

```
result += w[0]*d[0];
result += w[1]*d[1];
result += w[2]*d[2];
...
result += w[7]*d[7];
```

(a) Original code

```
v1 = w[0:3]*d[0:3]+w[4:7]*d[4:7];
v2 = v1[0:1] + v1[2:3];
result += v2[0] + v2[1];
```

(b) Pseudocode of successful vectorization after applying the *'mem2reg,slp-vectorizer'* sequence to the original code.

```
  w0=w[0];d0=d[0];
- sext i16 w0 d0 to i32; //sign extension from i16 to i32
+ sext i16 w0 d0 to i64; //sign extension from i16 to i64
  tmp = w0 * d0;
- sext i32 tmp to i64; //sign extension from i32 to i64
  result += tmp;
...
```

(c) The difference by applying *instcombine* after *mem2reg*.

Figure 5.1: An example from `telecom_gsm` in cBench showing how the phase order matters. Applying the *'mem2reg,slp-vectorizer'* pass sequence leads to successful vectorization, whereas *'mem2reg,instcombine,slp-vectorizer'* fails.

on an ARM multi-core CPU and a multi-core AMD x86 CPU. We compare CITROEN against state-of-the-art evolutionary algorithms and BO methods designed for program autotuning. Experimental results show that CITROEN outperforms these competitive baselines, achieving comparable tuning results with just one-third of their search budget. CITROEN proves especially effective with a constrained search budget - with a budget of 100 runtime measurements, it delivers up to a 17% improvement over random search and up to 10% over the strongest baseline.

5.2 Motivation

As a motivation example, consider applying phase ordering to LLVM v17.0 to optimise `telecom_gsm` from the cBench benchmark suite [162] on an ARM Cortex-A57 CPU on a Jetson TX2 platform. For this benchmark, the `long_term` module (i.e., `long_term.c`) contributes to more than 50% of the overall program execution time and is the target for optimisation here.

Figure 5.1a shows a hot code snippet that computes the dot product of two vectors, which would benefit from superword-level parallelism (SLP) vectorisation. Sequentially

Table 5.1: Applying different pass sequences to the `long_term` module in the `telecom_gsm` benchmark. By examining the relationship between pass-related compilation statistics and speedup (over -O3) from the first three samples, we can predict the fifth sample is more likely to be more profitable than the fourth sample.

No.	**Pass Sequence**	**Pass-related Compilation Statistics**				**Speedup**
		SLP.NVI	**mem2reg.NPI**	**mem2reg.NP**	**instcombine.NC**	
1	*mem2reg slp-vectorizer*	**14**	21	43	0	**1.13×**
2	*slp-vectorizer mem2reg*	0	21	43	0	0.85×
3	*inst-combine mem2reg slp-vectorizer*	0	18	41	271	0.85×
4	*mem2reg inst-combine slp-vectorizer*	0	21	43	244	0.86×
5	*mem2reg slp-vectorizer instcombine*	**14**	21	43	164	**1.14×**

applying the *mem2reg* and *slp-vectorizer* passes results in successful vectorisation, as shown in Figure 5.1b. However, when the *instcombine* pass is applied between *mem2reg* and *slp-vectorizer* (i.e., in the order of *'mem2reg, instcombine, slp-vectorizer'*), vectorisation fails due to the profitability analysis. This is because *instcombine* optimises greedily without considering the later vectorisation opportunity. Specifically, as can be seen from Figure 5.1c, *instcombine* reduces sign extension operations by converting an i16 to the i64 sign extension, but the resulting i64 instructions and data types are considered to be not profitable for applying vectorised horizontal reduction. Consequently, the LLVM vectoriser skips vectorisation on this code, leading to a performance slowdown compared to "-O3". If we can capture the interactions and the impact between compiler passes, we can then speed up phase ordering by avoiding profiling compiler sequences that are likely to offer no performance gain. Based on our preliminary experiments, we observe that pass-related compilation statistics can help us capture the relationship between pass sequences and performance.

Table 5.1 lists LLVM compilation statistics for five different pass sequences, along with their runtime performance, using the -O3 optimisation level as a baseline, where SLP.NVI, mem2reg.NPI, mem2reg.NP, and instcombine.NC stand for *SLP.NumVectorInstructions*, *mem2reg.NumPHIInsert*, *mem2reg.NumPromoted*, and *instcombine.NumCombined*, respectively. These statistics can be gathered using the '`-stats -stats-json`' flags of LLVM '`opt`' tool. Assume that we are exploring compiler pass sequences, and the first three sequences have already been profiled to obtain their execution times. The search algorithm must now assess whether the 4th and 5th pass sequences will likely be profitable and warrant further profiling. By effectively modelling compilation statistics, we may be able to identify performance improvements. In this example, the *SLP.NumVectorInstructions* metric is positively correlated with performance gains. Since the compilation statistics for the 5th pass sequence show a similar *SLP.NumVectorInstructions* value to that of the first sequence, which achieved a 1.13× speedup; this suggests that the 5th sequence is also likely to improve performance.

This example shows that pass-related compilation statistics can provide valuable insights, avoiding unnecessary profiling measurements to save search time. This motivates the design of a new search algorithm to leverage compilation statistics for phase ordering. By parallelising the compilation process to collect statistics, we can identify the most promising binaries for isolated runtime measurements, thereby reducing profiling overhead - a major bottleneck in compiler autotuning.

But how do we correlate compilation statistics with performance to model the interactions of compiler passes? A natural approach is to develop a model that maps compilation statistics to performance, which can serve as a utility function to guide the search. Since the correlation between compilation statistics and performance highly depends on the input program, we aim to train this model iteratively during the autotuning process, refining it as more profiling data is collected. For programs consisting of multiple source files (modules), it is also important to allocate the overall search budget across modules effectively to maximise the overall performance. Citroen is designed to address these challenges using BO as a search technique, as described in the next subsection.

5.3 Our Approach

5.3.1 Overview

Figure 5.2 depicts the workflow of Citroen. At the core of Citroen is a BO search component based on compilation statistics (Sec. 5.3.2), which will interact with a user-defined task function (Sec. 5.3.6) that defines how to compile and measure the generated binary. Citroen focuses on tuning "hot" modules whose accumulated execution time contributes to at least 90% of the overall program execution time. As a one-off profiling stage, Citroen identifies hot modules by using the Linux `perf` tool to profile the program compiled with the standard "-O3" optimisation flag. Specifically, we use

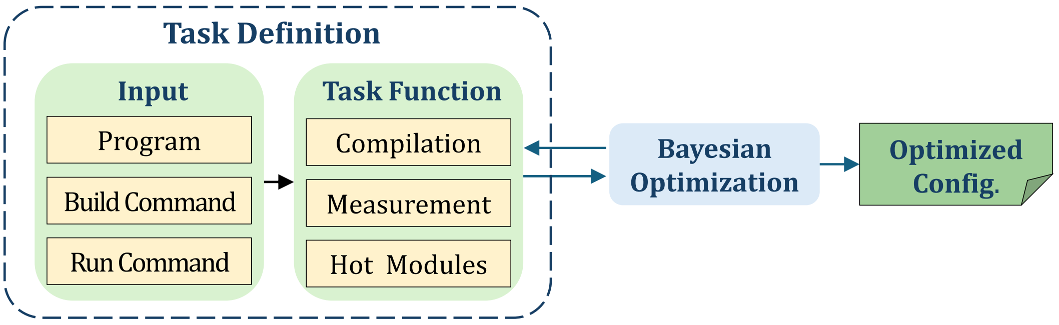


Figure 5.2: Overview of the CITROEN framework.

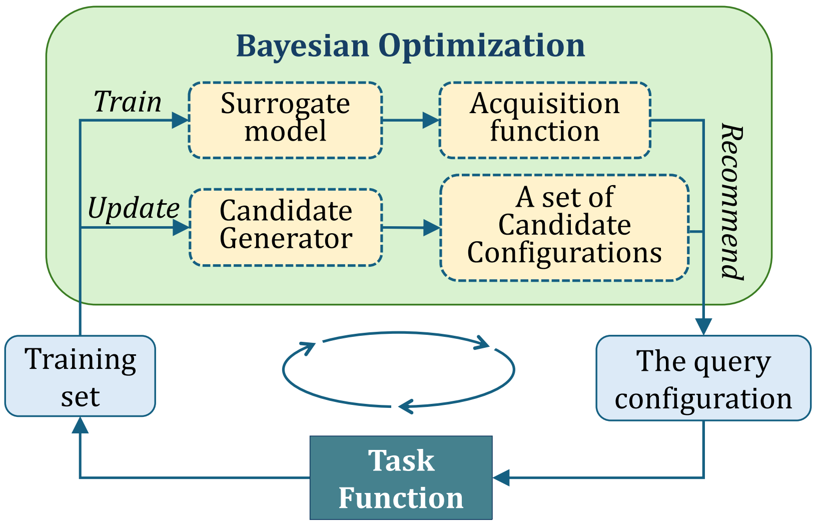


Figure 5.3: CITROEN's Bayesian optimisation workflow.

`perf` to measure the runtime of individual functions, excluding external calls, and then aggregate the execution times of functions within each source file to determine the hot modules. These identified hot modules are iteratively compiled with different pass sequences, while the remaining modules are compiled using -O3.

5.3.2 Bayesian Optimisation for Compiler Phase Ordering

Figure 5.3 outlines the workflow of CITROEN's BO component. We enhance standard BO with an online-trained cost model based on pass-related compilation statistics (Sec. 5.3.3), an acquisition function for navigating the non-uniform, sparse feature space (Sec. 5.3.4), and a heuristic pass sequence generator (Sec. 5.3.5). Instead of running separate BO processes for each source file, CITROEN fits a global cost model to estimate

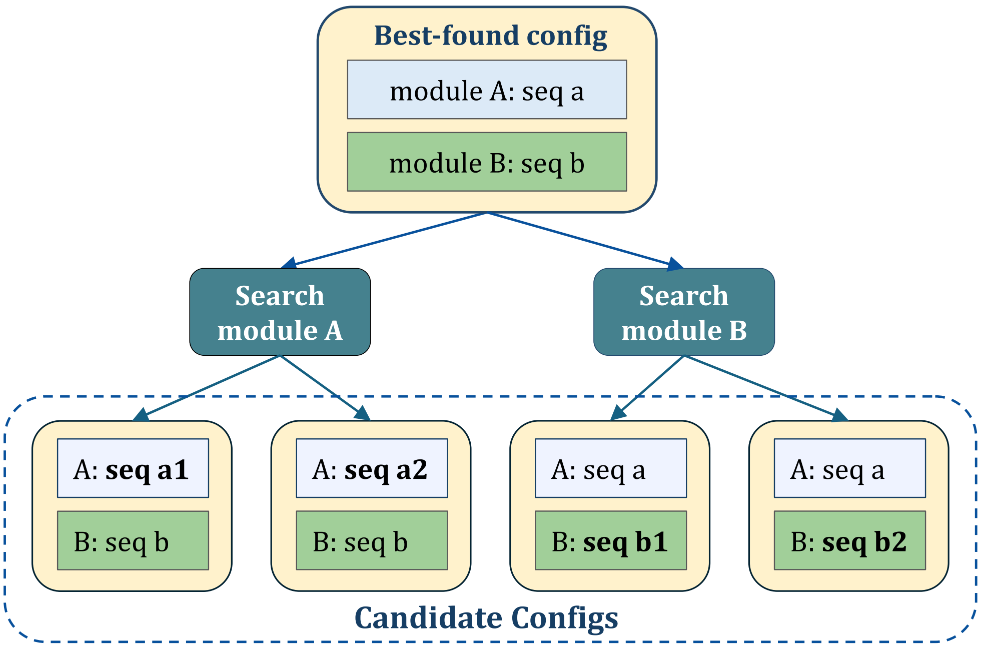


Figure 5.4: CITROEN's candidate configuration generator.

the impact of replacing the pass sequence of an individual module on overall program performance, dynamically determining which module to optimise while keeping others fixed.

In each iteration, CITROEN first learns a cost model that maps the compilation statistics of all hot modules to performance metrics (e.g., speedup over -O3). It then constructs an acquisition function to balance exploitation (prediction) and exploration (uncertainty). It also employs a candidate generator to produce pass sequences, which are then compiled in parallel to collect their statistics. As shown in Figure 5.4, CITROEN explores variations around the current best configuration. For each module, it proposes q new pass sequences while keeping the other modules fixed, resulting in $m \times q$ candidate configurations per iteration. Finally, the acquisition function selects the highest-value configuration to profile to obtain the execution time, which is then used to update both the cost model and the candidate generator.

For a single hot module, the acquisition function evaluates pass sequences within that module. For multiple hot modules, it decides which module to optimise next, allowing dynamic switching to maximise performance gains.

In particular, CITROEN's workflow differs from the standard BO workflow (Figure 2.1 in Section 2.1) in that it avoids the iterative search process typically used to maximise the acquisition function. Instead, it uses heuristics to generate candidate configurations without further search. This is because evaluating the acquisition function requires compilation, and directly generating candidate points allows for parallel compilation, thereby reducing the algorithm's overhead. This implementation is equivalent to the AIBO-none method described in Chapter 4, which also shows promising results in high-dimensional problems.

5.3.3 Cost Model for Performance Estimation

BO constructs a cost model (or surrogate model) to be used as a utility function to approximate the objective function. Prior work in BO-based compiler tuning [61, 143] uses the raw tuning parameters (e.g., compiler passes) as the cost model's input to predict the speedup or execution time. We take a different approach by using pass-related compilation statistics as the cost model's input.

We train and use the cost model following the standard 3-step of supervised learning: (1) feature extraction, (2) training and (3) inference, described as follows.

Feature extraction

Our cost model uses a feature vector of numerical values extracted from compilation statistics. The compilation statistics are gathered by adding the '`-stats -stats-json`' flags when using LLVM's `opt` tool to customise the pass sequence for a given module. After excluding statistics unrelated to performance optimisation (e.g., statistics of analysis pass), we are left with up to 255 statistics (usually fewer than 30), depending on the pass sequence and input program. We normalise the integer value of each statistic category to a range between 0 and 1 by dividing by its maximum observed value, forming a 255-dimensional feature vector (with most values being 0, as inactive passes generate no statistics). For programs with multiple hot modules, we form a single feature vector by concatenating the per-module feature vectors in a fixed module order to represent the entire program.

Training

CITROEN adopts the Gaussian process using the Matérn-5/2 kernel to build the cost model because it is proven to be effective in prior BO applications [10, 143]. The kernel function (which describes the similarity between two inputs) of this model is defined by

$$k(\mathbf{x}, \mathbf{x}') = \left(1 + \sqrt{5}d + 5d^2\right) e^{-\sqrt{5}d} \tag{5.1}$$

$$d = \sqrt{\sum_{i=1}^{D} \frac{(x_i - x_i')^2}{l_i^2}} \tag{5.2}$$

where d denotes the weighted Euclidean distance between two input feature vectors $\mathbf{x}$ and $\mathbf{x}'$. Here lengthscales l_i are hyperparameters that reflect the impact of each feature dimension on performance, which will be learned by minimising the negative log marginal likelihood loss function [22]. Initially, Citroen randomly generates n pass sequences, collecting their compilation statistics and evaluating the corresponding speedup over -O3 to construct the initial training set. In each subsequent iteration, Citroen will add the new evaluated sample to the training set to update the model.

Inference

Given a pass configuration c, we obtain its feature vector $\mathbf{x} = \varphi(c)$ by applying c to the input program, collecting compilation statistics, and normalising them into numerical feature values. Using the feature vector as input, the GP produces the prediction mean $\mu(\mathbf{x})$ and the prediction variance $\sigma^2(\mathbf{x})$, given by

$$\mu(\mathbf{x}) = K(\mathbf{X}, \mathbf{x})^T K(\mathbf{X}, \mathbf{X})^{-1} \boldsymbol{y} \tag{5.3}$$

$$\sigma^2(\mathbf{x}) = k(\mathbf{x}, \mathbf{x}) - K(\mathbf{X}, \mathbf{x})^T K(\mathbf{X}, \mathbf{X})^{-1} K(\mathbf{X}, \mathbf{x}) \tag{5.4}$$

where $\mathbf{X}$ denotes a set of training inputs, and $\boldsymbol{y}$ denotes the corresponding training labels (speedup over -O3). $K(\mathbf{X}, \mathbf{X})$ denotes the matrix containing all pairs of kernel entries, i.e. $K(\mathbf{X}, \mathbf{X})_{i,j} = k(\mathbf{x}_i, \mathbf{x}_j)$. $K(\mathbf{X}, \mathbf{x})$ denotes kernel values between training points and a test point, e.g., $K(\mathbf{X}, \mathbf{x})_i = k(\mathbf{x}_i, \mathbf{x})$. The mean and variance will then be used to construct an acquisition function to determine which compilation configuration to profile next.

5.3.4 Acquisition Function Design

Coverage issue. Citroen fits the cost model in a non-uniform, sparse feature space where many statistical categories contain zero values for a given pass sequence. This

Table 5.2: Applying 2,000 random pass sequences to different programs in cBench to observe whether randomly selected initial training sets can cover the feature space.

Initial training set size	**20**	**50**	**100**
Unexplored feature count (range)	$2 \sim 35$	$1 \sim 22$	$1 \sim 19$

creates a coverage issue in the initial training set. Unlike standard BO, where a uniform input space ensures effective coverage of each parameter dimension, CITROEN cannot directly sample in the feature space. As a result, generated candidate samples may include non-zero statistic categories not seen in the training set. To illustrate this point, we applied 2,000 random pass sequences to cBench programs and selected 20, 50, and 100 sequences per module for the initial training set. As shown in Table 5.2, the training set fails to cover all statistical categories, limiting the cost model's ability to predict configurations with unexplored features. This coverage issue should be addressed by designing the acquisition function to prioritise configurations with unexplored features.

An acquisition function is used to determine the next compilation configuration to be profiled by considering the trade-off between sampling from areas with better-predicted values $\mu(\mathbf{x})$ and exploring regions of high model uncertainty $\sigma(\mathbf{x})$. Standard BO usually uses expected improvement (EI) [19] as the acquisition function, defined by

$$\mathrm{EI}(\mathbf{x}) = \mathbb{E}[\max(f^* - y, 0) \mid y \sim \mathcal{N}(\mu(\mathbf{x}), \sigma^2(\mathbf{x}))]$$

where f^* is the best function value observed so far.

However, because of the coverage issue, directly using acquisition functions designed for standard BO for CITROEN can lead to inadequate exploration. Although the EI function encourages the exploration of configurations with high model uncertainty, the model uncertainty itself is inaccurate for configurations with unexplored features because it does not consider the effect of unexplored features. As we will show in Sec. 5.5.2, using standard EI for CITROEN could lead to suboptimal performance.

To tackle the issue, we develop a customised acquisition function $\alpha(\mathbf{x})$ for the compiler phase-ordering problem based on EI, defined by

$$\alpha(\mathbf{x}) = \begin{cases} \mathrm{EI}(\mathbf{x}) + \widetilde{\lambda}, & \text{OOD} \\ \mathrm{EI}(\mathbf{x}), & \text{not OOD} \end{cases} \tag{5.5}$$

where OOD means out of distribution, i.e., the test candidate point $\mathbf{x}$ owns a specific non-zero feature (compilation statistic) which is not included in the training set. To encourage the exploration of configurations with such unseen features, we add a large constant $\widetilde{\lambda}$ to the EI value of OOD candidates so that they receive higher acquisition function values and are prioritised during selection. In our experiments, we set $\widetilde{\lambda} = 10^8$, which is sufficiently large to dominate the typical EI values; the exact value is not critical as long as it is large enough to ensure prioritisation.

5.3.5 Pass Sequence Generator

Due to the large number of possible pass sequences, it is impossible to evaluate the acquisition function values of all sequences. Based on the heuristic acquisition function initialisation scheme for high-dimensional BO proposed in Chapter 4, Citroen employs two heuristic sampling strategies to generate candidate samples that are more likely to have better performance. While Chapter 4 employs GA and CMA-ES for continuous optimisation problems, Citroen uses GA and Discrete 1+1 Evolution Strategy (DES) for generating sequences consisting of categorical variables (passes).

GA-based Sampling

Specifically, Citroen maintains 20 top-performing pass sequences for each module as the GA population and applies selection, crossover and mutation operations to this population to generate candidate offspring sequences.

Selection. Citroen adopts the same tournament selection strategy described in Section 4.3.2 of Chapter 4.

Crossover. Citroen combines two parent pass sequences to implement a one-point crossover, i.e., selecting a single crossover point on each parent sequence and swapping the segments from that point onward to generate new sequences. The crossover probability is set to 0.9.

Mutation. Citroen uses random mutation to change a small part of the passes in a parent pass sequence. The mutation probability is set to 1/D, where D is the dimension of the problem, i.e., the length of the pass sequence.

DES-based Sampling

Unlike GA-based sampling, the DES-based sampling strategy only maintains the best-performing pass sequence. At each iteration, this kind of sampling involves randomly replacing a certain percentage of passes in the best-found pass sequence to generate candidate sequences. Specifically, it applies random replacements to 10%, 20%, 50%, and 100% of the passes in the sequence, with each proportion having an equal probability.

5.3.6 Autotuning Task Definition

To implement phase order autotuning, users must define a task function that compiles a program with a specific configuration and measures the performance of the resulting binary. Compilers like LLVM do not allow direct phase order specification per module, so previous frameworks [62, 67, 143] require users to manually re-implement the compilation process for different pass sequences, which requires engineering efforts, especially for programs with multiple source files and complex compilation processes. Citroen simplifies this by automating the task function definition without manual re-implementation. As shown in Figure 5.5, Citroen leverages the program's existing build script (e.g., makefile) and uses *clangopt* (line 10) as the compiler. Unlike LLVM's clang, *clangopt* reads the compilation configuration from a JSON file before invoking LLVM to handle the customised compilation. Running the build script with *clangopt* automates the execution of multiple compilation commands, each using the specified configurations. Additionally, Citroen includes features like automatic hot module detection (line 19) and remote execution support (line 16), reducing the engineering effort required for phase order tuning.

5.4 Experimental Setup

5.4.1 Implementation

We implemented Citroen in around 5K lines of Python code. We use the GPyTorch [22] GP library to implement the GP regression process as the cost model of BO.

5.4.2 Evaluation Platforms

Hardware platforms. We execute the search algorithm of Citroen on a multi-core server powered by two 20-core Intel Xeon Gold 5218R CPUs. Citroen then cross-compiles binaries on the host machine and sends the compiled binaries for execution

```python
import citroen
from citroen.function_wrap import Function_wrap
from citroen.utils import gen_hotfiles
from citroen.BO.BO import BO
from fabric import Connection

# Define the task function
fun = Function_wrap(
    # Explicitly declare the compiler as clangopt
    build_cmd='make CC=clangopt',
    build_dir='Example',
    # User-defined run and evaluation command
    run_and_eval_cmd='./run_eval.sh',
    binary_name='a.out',
    remote_run_dir='home/usr/RemoteExample',
    ssh_connection=Connection(host="xxx.xxx.xxx")
    )

# Automatically recognise hotfiles
hotfiles = gen_hotfiles(fun)
fun.hotfiles = hotfiles

# Autotuning the phase order of the program
optimizer = BO(fun=fun, budget=1000)
best_cfg, best_cost = optimizer.minimize()
```

Figure 5.5: An example of using CITROEN for phase ordering.

Table 5.3: LLVM optimisation passes considered in evaluation

adce, aggressive-instcombine, alignment-from-assumptions, annotation2metadata, argpromotion, bdce, called-value-propagation, callsite-splitting, cg-profile, chr, constmerge, constraint-elimination, coro-cleanup, coro-early, coro-elide, coro-split, correlated-propagation, deadargelim, div-rem-pairs, dse, early-cse, elim-avail-extern, float2int, forceattrs, function-attrs, globaldce, globalopt, gvn, indvars, inferattrs, inject-tli-mappings, inline, instcombine, instsimplify, ipsccp, jump-threading, libcalls-shrinkwrap, licm, loop-deletion, loop-distribute, loop-idiom, loop-instsimplify, loop-load-elim, loop-rotate, loop-simplifycfg, loop-sink, loop-unroll, loop-unroll-full, loop-vectorise, lower-constant-intrinsics, lower-expect, mem2reg, memcpyopt, mldst-motion, move-auto-init, openmp-opt, openmp-opt-cgscc, reassociate, rel-lookup-table-converter, rpo-function-attrs, sccp, loop-unswitch, simplifycfg, slp-vectorizer, speculative-execution, sroa, tailcallelim, vector-combine, break-crit-edges, loop-data-prefetch, loop-fusion, loop-interchange, loop-unroll-and-jam, lowerinvoke, sink, ee-instrument

and performance measurement on two platforms: an ARM-based NVIDIA Jetson TX2 board with a 64-bit quad-core ARM Cortex A57 running at 2.0 GHz and a multi-core server with a 64-core AMD Ryzen Threadripper PRO 5995WX CPU clocked at 2.25 GHz. The benchmarks are run as single-threaded programs on the CPU. The SPEC CPU 2017 benchmarks are evaluated solely on the x86 platform due to their long execution time on the Jetson TX2 board.

Compiler. We apply Citroen to LLVM version 17.0.6. Our evaluation considers 76 LLVM passes listed in Table 5.3 and a maximum compiler sequence of 120 passes.

5.4.3 Benchmarks

Table 5.4 lists the benchmarks used in the experiments, including 26 programs from cBench [162] and 16 programs from SPEC CPU 2017 [16]. We only consider C/C++ programs that can be successfully compiled by LLVM v17.

5.4.4 Competing Baselines

We compare Citroen against five autotuning methods and alternative feature extraction methods:

Random. While simple, random search is reported to be effective in previous work [52, 71, 72].

OpenTuner. This compiler auto-tuning framework [62] implements an ensemble of multiple evolutionary algorithms and can dynamically adjust its use of different algorithms.

Nevergrad. This search library [95] supports multiple evolutionary algorithms. It could adaptively select the most suitable algorithm according to the search problem setting. This method has been reported to achieve the best performance in the CompilerGym [67] phase-ordering environment.

BOCA. This closely related work uses BO for *compiler flag selection* [61]. It uses the random forest as its cost (surrogate) model. When applying it to phase ordering, we adapt it to use one-hot encoding as the input to the random forest model.

BaCO. This is a BO framework for compilation optimization [143]. It can handle different parameter types and thus can be directly used for the compiler phase-ordering problem.

Table 5.4: Benchmarks used in evaluation.

Suite	ID	Benchmark	#hot modules
cBench [162] (budget: 100/300/1000, platform: ARM and x86)	C1	automotive_bitcount	4
	C2	automotive_qsort1	2
	C3	automotive_susan_c	1
	C4	automotive_susan_e	1
	C5	automotive_susan_s	1
	C6	bzip2d	2
	C7	bzip2e	3
	C8	consumer_jpeg_c	6
	C9	consumer_jpeg_d	4
	C10	consumer_lame	8
	C11	consumer_tiff2bw	3
	C12	consumer_tiff2rgba	3
	C13	consumer_tiffdither	3
	C14	consumer_tiffmedian	1
	C15	network_dijkstra	1
	C16	network_patricia	1
	C17	office_stringsearch1	1
	C18	security_blowfish_d	2
	C19	security_blowfish_e	2
	C20	security_rijndael_d	1
	C21	security_rijndael_e	1
	C22	security_sha	1
	C23	telecom_CRC32	1
	C24	telecom_adpcm_c	1
	C25	telecom_adpcm_d	1
	C26	telecom_gsm	5
SPEC CPU 2017 [16] (budget: 100/300, platform: x86)	S1	500.perlbench_r	6
	S2	502.gcc_r	7
	S3	505.mcf_r	3
	S4	508.namd_r	2
	S5	510.parest_r	6
	S6	511.povray_r	9
	S7	519.lbm_r	1
	S8	520.omnetpp_r	9
	S9	523.xalancbmk_r	9
	S10	525.x264_r	6
	S11	526.blender_r	3
	S12	531.deepsjeng_r	9
	S13	538.imagick_r	1
	S14	541.leela_r	4
	S15	544.nab_r	2
	S16	557.xz_r	5

Feature extraction methods. CITROEN uses compilation statistics as features to be given to the BO cost model to predict potential speedup and uncertainty. In Sec. 5.5.3, we compare CITROEN against three feature extraction methods: IR2vec [149], Autophase [146], and Programl [150].

5.4.5 Evaluation Methodology

Hyper-parameters of Citroen. Based on our preliminary experiments, in our experiments, we set the initial training samples for the cost model (n_init) to 20 and the candidate pass sequences per iteration (q) to 500. All candidate sequences are initially generated using the heuristic sampling strategy described in Sec. 5.3.5. After 1/4 of the total search iterations, CITROEN generates 50 new sequences per module, with the remaining $q-50$ selected randomly from previously generated but unevaluated sequences, keeping compilation overhead negligible compared to execution overhead.

Compiling multiple modules. To apply the competing baselines (Sec. 5.4.4) to optimise module-specific phase ordering of programs with multiple source files, we use a one-by-one strategy to sequentially auto-tune each module in descending order of their execution times. We tune each module until there is no noticeable performance improvement (more than 1% speedup) for τ consecutive search iterations before moving to the next one. Here, τ is set to $N\_budget/N\_modules/3$. We will repeat the process until the search budget is used up. When re-tuning a module, we initialise the search algorithm using the best-found sample from the search history. In this way, these baselines will not waste too much time on source files and will have little room for performance improvement.

Performance report. Following [66, 75], we set search budgets of 100, 300, and 1000 iterations for cBench and 100 and 300 iterations for SPEC CPU 2017, with the latter capped at 300 due to long execution times. In each iteration, we execute the compiled binary multiple times until the relative standard error of the mean execution time falls below 1% (typically requiring 3-20 runs for cBench and 3 for SPEC). The mean execution time is then used as feedback for the search algorithm. When reporting the final performance, we re-execute the best-found binary until the relative standard error falls below 0.3% for greater accuracy. For each method, we report the average performance by repeating the tuning process five times per benchmark.

5.5 Experimental Results

Our evaluation tries to answer the following questions:

RQ1: How does CITROEN compare with prior autotuning approaches (Sec. 5.5.1)?

RQ2: How do individual components of CITROEN contribute to its overall performance (Sec. 5.5.2)?

RQ3: How do CITROEN's pass-related compilation statistics compare with existing feature extraction methods (Sec. 5.5.3)?

5.5.1 Comparison with Baselines

Figure 5.6 shows the average performance of CITROEN and the baselines with three different budgets on cBench and SPEC CPU 2017. CITROEN clearly outperforms the baselines by both achieving the same performance faster and achieving better performance on a small budget (e.g., 100 iterations). For cBench, with a small budget of 100 iterations, CITROEN achieves 1.096× speedup over -O3 compared to other methods' 1.067 – 1.083× speedup. With a moderate budget of 300 iterations, CITROEN attains a 1.11× speedup, which other methods require 1000 iterations to match.

To see how CITROEN generalises across different benchmarks, we show in Figure 5.7 the comparison of CITROEN and the baselines on individual benchmarks. The results show that CITROEN significantly improved performance on several benchmarks, like C1, C22 and C26. The common characteristic of these benchmarks is that their performance can benefit from some transformations, but the compilation sequences that can activate these transformations are sparse in the search space. For example, for C1 (`automotive_bitcount`), to achieve more than 1.1× speedup, one of its hot modules `bitcnts` should be optimised by three loop transformations including *loop-unswitch*, *loop-unroll* and *licm*. However, all the baselines struggle to suggest a good compilation pass sequence that activates all three transformations within a budget of 100 profiling measurements. Furthermore, for C22, we discovered that the combination of *early-cse*, *instcombine*, *loop-rotate*, and *loop-fusion* passes successfully unlocks loop-level optimisation. For S10, the key is to apply *loop-unroll* before and after the *instcombine* pass to enhance instruction-level parallelism. For S11, we improve vectorisation by identifying a sub-sequence that applies *slp-vectorizer* after *sroa* and *simplifycfg*.

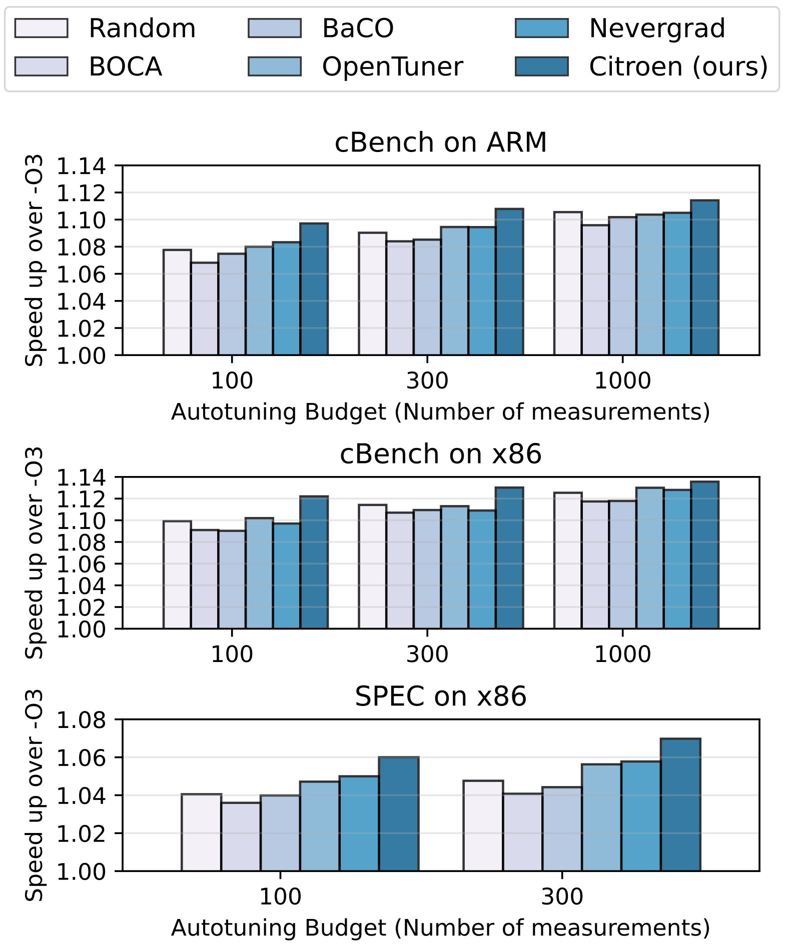


Figure 5.6: Average (geometric) performance on cBench and SPEC CPU 2017 with different search iteration budgets.

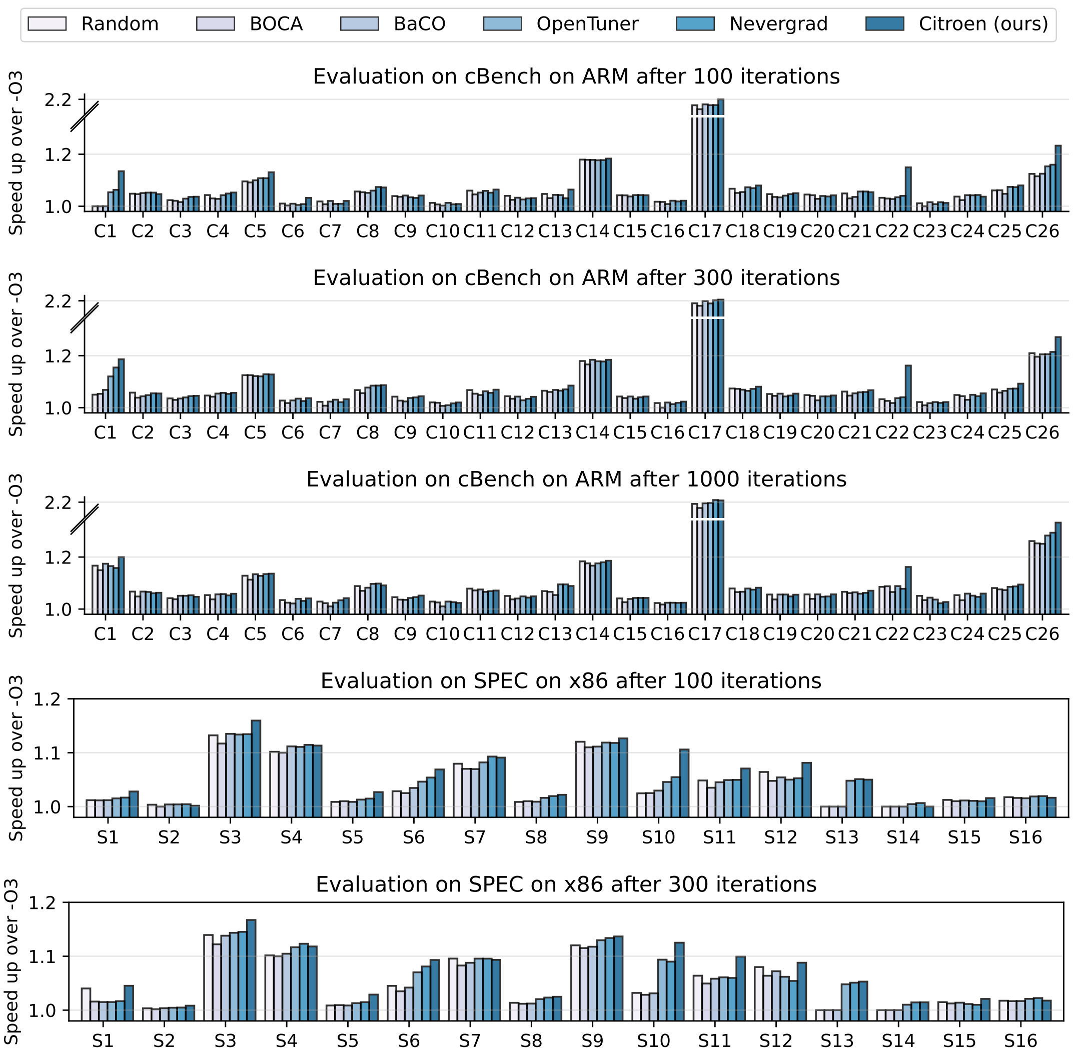


Figure 5.7: Evaluation on cBench and SPEC with different search iteration budgets.

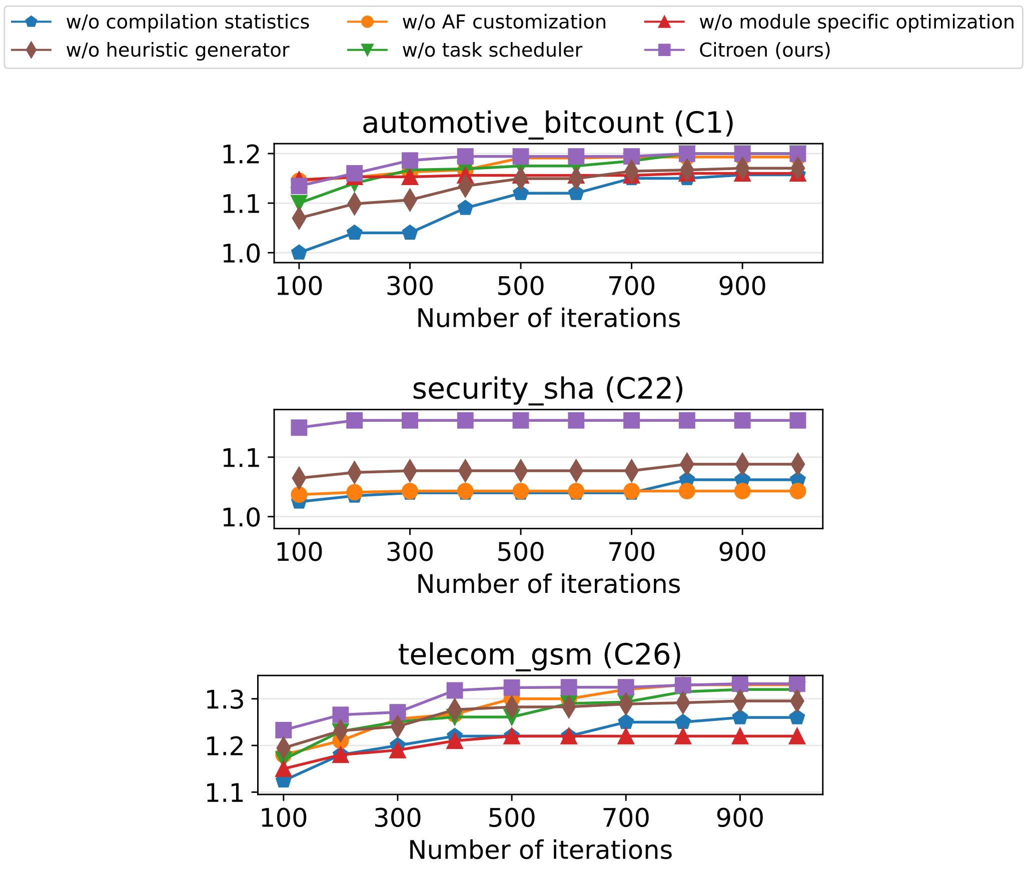


Figure 5.8: Ablation study on different benchmarks. The y-axis is the speedup relative to -O3. `security_sha` only owns one hot module, thus “task scheduler” and “module specific optimisation” are not applicable to such single-module cases.

Another observation is that many benchmarks allow all the methods to achieve similar performance. These benchmarks share the common characteristic that their performance under a small search budget (100) is close to that achieved with a large budget (1000). This is because their performance is dominated by easily activated optimisations (like *mem2reg*). This observation is also consistent with previous studies [52, 71, 72], where random search is reported to be effective enough in many cases.

5.5.2 Ablation Study

To assess how each component of CITROEN impacts performance, we evaluate its variants on the ARM platform on several cBench benchmarks. The "CITROEN (ours)" variant uses all proposed techniques. "W/o compilation statistics" uses original pass sequences instead of compilation statistics for the cost model. "W/o heuristic generator" utilises random sampling rather than heuristic algorithms to generate candidate pass sequences. "W/o AF customisation" employs standard EI as the acquisition function, ignoring the coverage issue. "W/o task scheduler" sequentially autotunes each module instead of using a global task scheduler. "W/o module-specific optimisation" applies a single pass sequence for all modules.

As can be seen in Figure 5.8, "W/o compilation statistics" performs much worse than CITROEN in terms of both the final achieved performance and search efficiency, indicating that pass-related compilation statistics are a key component of CITROEN. "W/o heuristic generator" outperforms "W/o compilation statistics" but remains less effective than CITROEN. This is because it lacks an efficient initialisation strategy for the acquisition function maximisation process, which Chapter 4 highlights as crucial for high-dimensional problems. "W/o AF customisation" uses 1000 search iterations to achieve only 1.04× speedup in `security_sha` (C22) while CITROEN uses 100 iterations to achieve 1.16× speedup, showing that the coverage issue could significantly harm performance in some cases. For programs with multiple hot modules, "W/o module specific optimisation" performs the worst in terms of the final achieved performance, revealing the effectiveness of module-specific optimisation. As depicted in "W/o task scheduler", one-by-one autotuning could achieve module-specific optimisation to improve the final performance, but it requires more search iterations. This demonstrates how a global model could effectively act as a task scheduler to adaptively allocate search budgets when autotuning programs with multiple hot modules.

5.5.3 Alternative Feature Extraction Methods

Prior works in machine learning-based compiler optimisation developed a range of methods to extract features from intermediate representations (IRs) to train offline supervised or reinforcement learning models for predicting optimal compilation configurations.

IR2vec [149], Autophase [146], and Programl [150] provides three representative

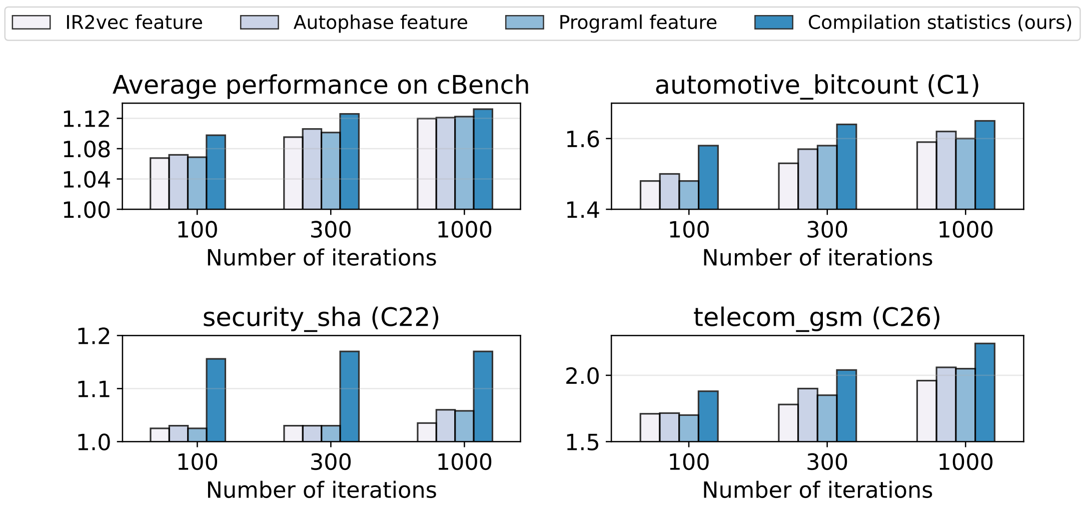


Figure 5.9: Impact of replacing compilation statistics with alternative feature extraction methods in Citroen (using LLVM 10 as the compiler). The y-axis shows the speedup relative to -O3.

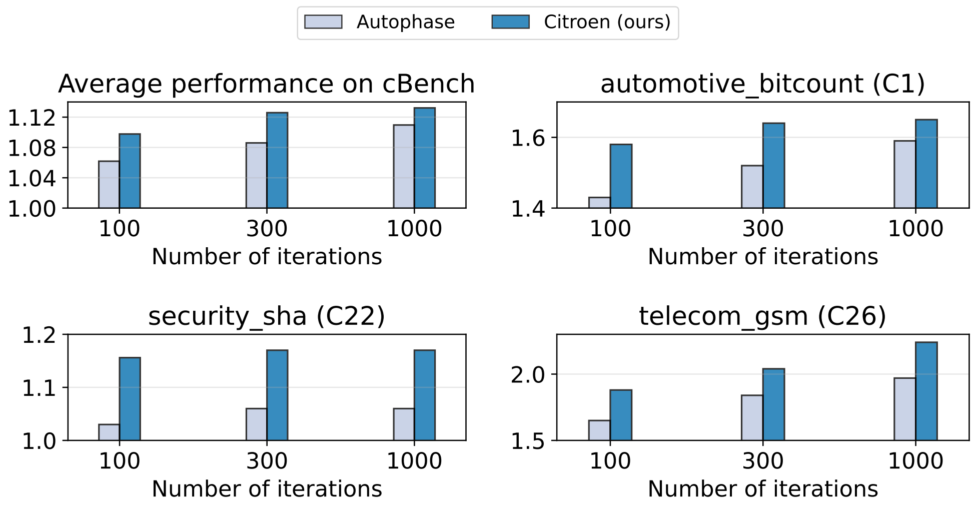


Figure 5.10: Comparison of Citroen and Autophase using LLVM 10 as the compiler. The y-axis is the speedup relative to -O3.

feature extraction techniques. IR2vec combines representation learning with control flow information to embed IRs in a continuous space. Autophase extracts static features via analysis passes on IRs. Programl represents programs as graphs to capture their semantics and employs inst2vec [148] for continuous embeddings. Although these methods are not tailored for search-based autotuning, their feature extraction techniques could be integrated into our approach to construct an online cost model and thus warrant comparison.

Figure 5.9 shows how our compilation statistics perform on the arm platform compared to alternative feature extraction methods, including IR2vec [149], Autophase [146], and Programl [150]. As both Autophase and Programl only support LLVM 10, here we use LLVM 10 to compare all feature extraction methods fairly. Using pass-related compilation statistics as features, CITROEN clearly outperforms other feature extraction methods. This is because alternative feature extraction methods struggle to distinguish the changes brought by various passes. For example, the *function-attrs* pass could significantly affect the performance of some programs like `automotive_bitcount`, but its transformation on the program can not be recognised by IR2vec, Autophase and Programl, as *function-attrs* only changes the function attributes which are not considered in those feature extraction methods.

Furthermore, while IR2vec and Programl do not generate or suggest compiler pass sequences and must be integrated with a separate phase ordering method, Autophase provides both a feature extraction mechanism and an end-to-end reinforcement learning (RL)-based phase ordering solution. Thus, we also compare CITROEN directly with Autophase as a complete solution. We explored both offline and online approaches in Autophase. First, we trained a proximal policy optimisation (PPO) model on 100 randomly generated programs from Csmith [170], following the approach in Autophase. Then, we used this model as the initial policy for further RL-based search. Figure 5.10 reports the results, where CITROEN still consistently outperforms Autophase.

5.5.4 Hyperparameter Sensitivity Analysis

CITROEN has two key hyperparameters: the initial training samples for the cost model (n_init) and the candidate pass sequences per iteration (q). Figure 5.11 reports how different hyperparameter values affect CITROEN's average performance across cBench benchmarks on the ARM platform. CITROEN demonstrates overall robustness to dif-

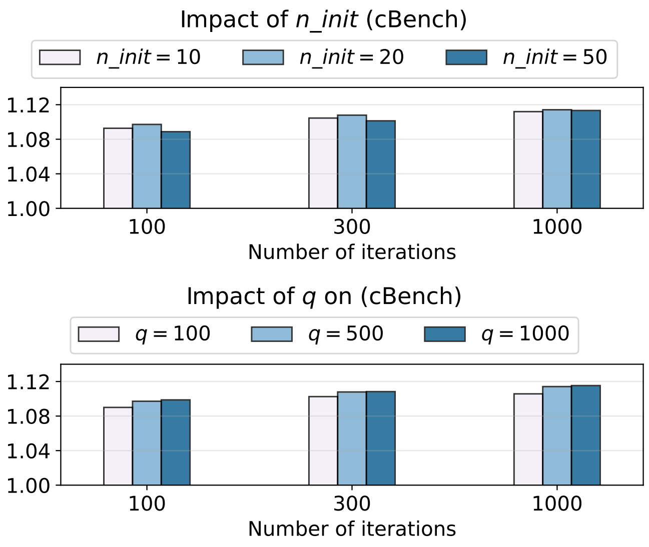


Figure 5.11: Hyperparameter Sensitivity Analysis of CITROEN. The y-axis is the speedup relative to -O3.

ferent hyperparameter values, except that too small q may lead to marginally degraded performance. Furthermore, increasing q beyond 500 does not result in any substantial improvement in performance. Additionally, when the proportion of *n_init* relative to the total number of search iterations is large, it can cause a slight degradation in performance. This is because a higher proportion of the search budget is allocated to random sampling, which may be less efficient.

5.5.5 Compilation Statistics Analysis

Table 5.5 presents an experiment designed to explore the relationship between various compilation statistics and the resulting performance speedup. For each program, we focus on the module with the longest runtime, running CITROEN for 1000 iterations to derive both the optimal pass sequence and the final cost model. Using the cost model's lengthscales l_i (as defined in equation 5.1), we identify impactful features (compilation statistics), where a smaller lengthscale indicates a greater influence on performance. To assess the significance of each feature, we measure the change in performance after removing the passes associated with that feature from the final pass sequence. For instance, if *loop-unroll.NumUnrolled* is identified as impactful by the cost model, we remove *loop-unroll* from the pass sequence to measure its effect on

Table 5.5: Top 5 impactful compilation statistics recognised by the CITROEN cost model on different benchmarks. Performance changes are measured after removing the relevant passes from the final pass sequence.

Benchmark	Compilation Statistics	Performance Change if Disabling Related Passes
`automotive_bitcount`	*loop-unroll.NumUnrolled*	-49%
	inline.NumInlined	-43%
	licm.NumHoisted	-54%
	mem2reg.NumPHIInsert	-52%
	loop-unswitch.NumBranches	-22%
`security_sha`	*instcombine.NumCombined*	-27%
	mem2reg.NumPHIInsert	-21%
	loop-rotate.NumRotated	-26%
	early-cse.NumCSE	-16%
	loop-unroll.NumUnrolled	-5%
`telecom_gsm`	*mem2reg.NumPHIInsert*	-31%
	SLP.NumVectorInstructions	-19%
	instcombine.NumCombined	-5%
	loop-vectorize.LoopsVectorized	-7%
	simplifycfg.NumSimpl	-5%

performance. The results reveal that the set of impactful compilation statistics varies across programs, indicating that different programs are influenced by distinct compilation factors. However, some statistics consistently emerge as impactful across multiple programs, highlighting their broader relevance in program optimisation.

Furthermore, to understand the correlation between statistics and pass sequences, we try to reduce the pass sequence to find the most important pass combination that will affect statistics. We found that some passes are naturally correlated and often appear together in sequences to make the occurrence of certain statistics possible. For example, *loop-vectorise.LoopsVectorized* usually at least requires the sequential application of both the *loop-rotate* and *loop-vectorise* passes. Similarly, applying *sroa* before *slp-vectorizer* often leads to larger *SLP.NumVectorInstructions*. Additionally, we found that in most cases, replacing *mem2reg* with *sroa* typically yields comparable *mem2reg.NumPHIInsert* values and similar runtime performance.

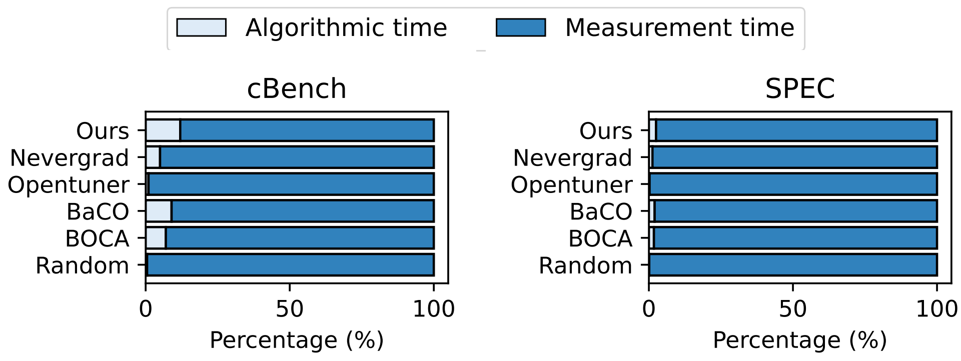


Figure 5.12: Average proportion of algorithmic runtime.

5.5.6 Algorithmic Runtime

In Figure 5.12, we report the average proportion of algorithmic runtime (excluding the time spent on objective function evaluation) across different methods for 1000 search iterations in cBench and 300 iterations in SPEC CPU 2017. For CITROEN, as it requires extra parallel compilation (only on hot modules) and model training and inference, it uses more algorithmic runtime than other methods. However, its algorithmic runtime remains negligible compared to the performance measurement time (including program compilation and execution time), especially when optimising larger programs like SPEC CPU 2017.

5.6 Summary

This chapter has presented CITROEN, a BO-based search framework for compiler phase ordering. While the last chapter addresses the high-dimensional search space challenge (Section 1.2.1), this chapter focuses on addressing the compiler pass interaction challenge (Section 1.2.2) and the practicality challenge (Section 1.2.3).

CITROEN leverages pass-related compilation statistics to capture the complex interactions between compiler passes, providing features that are more closely aligned with performance. This enables the construction of a more effective online probabilistic cost model, which helps avoid profiling pass sequences that yield no performance improvement. In this way, CITROEN addresses the challenge of compiler pass interaction.

To address the practicality challenge, CITROEN automates the task function definition without manual re-implementation, simplifying the autotuning process. It also implements a dynamic budget allocation across source files to support module-specific phase ordering, improving search efficiency. Our evaluation shows that CITROEN outperforms existing approaches by achieving comparable tuning results using one-third of their search budget.

Chapter 6

Conclusions

This chapter summarises the main contributions of this thesis in Section 6.1, presents a critical analysis of this work in Section 6.2 and discusses possible directions for future research in Section 6.3.

6.1 Contributions

This thesis presents new techniques for applying Bayesian optimisation (BO) to compiler autotuning. It especially focuses on one of the most challenging compiler autotuning problems, i.e., phase ordering. As a departure from previous BO techniques in software/compiler autotuning [141–143], it customises BO to address the challenges outlined in Section 1.2, including the high-dimensional optimisation space, the complex compiler pass interactions, and the practicality barrier.

6.1.1 A Simple and Effective High-Dimensional BO Method

Chapter 4 addresses the challenge of applying BO to high-dimensional optimisation problems (Section 1.2.1), which is a crucial step towards applying BO to the high-dimensional compiler phase ordering problem. It is the first work to investigate a largely ignored yet significant problem in high-dimensional BO concerning the impact of the initialisation of the acquisition function (AF) maximiser on the realisation of the AF capability. It empirically shows that the commonly used random initialisation strategy leads to poor-quality candidate points that limit AFs' power, leading to over-exploration and poor high-dimensional BO performance. Based on this understanding, it proposes a simple yet effective initialisation method by employing multiple heuristic optimisers to leverage the historical data of black-box optimisation to generate initial points for the AF maximiser.

The proposed method is simpler and more efficient than previous high-dimensional BO methods. The simplicity of the method makes it easy to implement and apply to various high-dimensional optimisation problems, including the compiler phase ordering problem. The method is also more efficient, requiring fewer evaluations to find better solutions. Especially while testing in a high-dimensional compiler flag selection problem, the proposed method outperforms the standard BO method by delivering a 7% runtime improvement after 1000 search iterations. It also outperforms other high-dimensional BO methods by delivering 2-6% improvement.

6.1.2 Customisation for Compiler Phase Ordering

Chapter 5 addresses the challenge of complex compiler pass interactions (Section 1.2.2), which complicates the task of predicting the performance impact of a given pass sequence. It leverages pass-related compilation statistics as a proxy to capture the compiler pass interactions. As pass-related compilation statistics directly reflect the changes made by each pass, they provide a more direct representation of the pass interactions than the pass sequence itself, avoiding the need to model the sequence. Compared to the standard BO, the proposed technique inserts a new layer of abstraction (compilation statistics) between the pass sequence and the performance, making the modelling process more straightforward and effective, but introducing a non-uniform, sparse statistics feature space. To better navigate the non-uniform, sparse statistics feature space, the proposed technique customises the acquisition function design by incorporating a coverage-based term. The coverage-based term encourages the model to explore the feature space more uniformly, leading to better performance. Besides, it also extends the heuristic initialisation strategy proposed in Chapter 4 to a discrete version to generate high-quality candidate pass sequences to be selected by the acquisition function.

Compared to previous BO techniques in performance autotuning [141–143], the proposed technique can achieve better tuning results with just one-third of their search budget. It proves especially effective with a constrained search budget - with a budget of 100 runtime measurements, it delivers up to a 17% improvement over other BO techniques.

6.1.3 Practical Multi-Module Autotuning

Chapter 5 also addresses the practicality barrier challenge (Section 1.2.3) by developing a dynamic budget allocation strategy and a user-friendly framework for multi-module compiler phase ordering.

The dynamic budget allocation strategy allocates the search budget across multiple source files within a single program based on the real-time autotuning results of each file. This strategy ensures that the next search iteration is allocated to the file with the highest expected improvement. This is achieved by implementing a global cost model that considers multiple modules' effects rather than a single-module cost model during the BO search process. Compared to the one-by-one autotuning process, the proposed dynamic budget allocation strategy can achieve up to a 2.5× faster convergence time.

The user-friendly framework provides a simple and efficient way to perform multi-module compiler phase ordering for real-life large applications. It allows users to directly use the program's existing build script (e.g., makefile) without rewriting the compilation process to accept a specific compilation configuration, thus reducing the user barrier.

6.2 Critical Analysis

6.2.1 Pass-Related Compilation Statistics

The pass-related compilation statistics used in this thesis are aggregated results, risking neglecting lower-level details. For instance, while the statistics can indicate the number of loops vectorised, they do not specify which loops were vectorised, their locations, or their significance in the overall performance. This limitation introduces inaccuracies in the modelling process, as the model cannot differentiate between more and less critical optimisations. Although our online modelling approach accounts for model uncertainty, the granularity of the statistics remains a challenge. Developing a more detailed feature extraction method might enhance the accuracy of the online predictive model and improve the overall effectiveness of the compiler autotuning process.

6.2.2 Program-Specific Autotuning

Our proposed BO approach searches from scratch for each targeting program, ensuring flexibility and adaptability to different scenarios. However, this method may overlook potential opportunities, such as the existence of general program-independent pass correlations. These correlations, if identified and leveraged, could enhance the efficiency of the optimisation process. By not utilising these potential correlations, our approach might miss out on optimisations that could be applied across different programs.

6.2.3 Additional Compilation and Modelling Cost

Our approach uses post-compilation features to predict performance online, inherently introducing additional compilation and modelling overhead. To mitigate this overhead, we employ parallel compilation techniques and restrict the compilation process to hotspot files, which are the files most likely to impact performance. This strategy significantly reduces the overall compilation cost. Additionally, we utilise the highly efficient GPyTorch library to leverage GPU resources for model training and inference. In practice, the additional overhead incurred is minimal when compared to the time required for runtime performance measurement. However, this approach requires the algorithm and compilation to be executed on high-performance multi-core servers to ensure efficiency.

6.2.4 Optimisation Objective

This thesis primarily targets execution time optimisation in compiler autotuning and does not explicitly address other objectives such as code size or energy consumption. Code size optimisation is mostly static and less challenging, while energy consumption is often correlated with execution time and would require more detailed hardware-level modelling. Focusing on runtime performance aligns with the main industrial interest, but limits the scope of the work to a single objective setting.

6.3 Future Work

6.3.1 Coverage-Based Code Characterization

This thesis uses pass-related compilation statistics to model the relationship between compiler pass sequences and runtime performance. These statistics reveal whether and how a specific pass changes the code. However, they do not provide detailed information about what code snippets are changed and their significance in the overall performance.

One approach to addressing the limitations of compilation statistics is to incorporate profiling-based code coverage, which only needs to be performed once. Profiling-based coverage provides detailed insights into the execution frequency of instructions within each basic block of the intermediate representation (IR). By extracting coverage-based features from IR, we can differentiate between critical and non-critical code segments. Since different pass sequences lead to different transformations, identifying which changes have a greater impact can improve predictive modelling.

Future work could focus on leveraging coverage-based features to refine the modelling process, enhancing the accuracy of performance predictions and enabling more effective compiler autotuning.

6.3.2 Exploiting Program-Independent Pass Correlations

Some passes are naturally correlated and often appear together in sequences. Considering these program-independent relationships might help reduce the search space and streamline the optimisation process. Future work could involve developing methods to identify and exploit these program-independent pass correlations. By incorporating these correlations into the optimisation process, we could potentially reduce the number of evaluations required to find optimal pass sequences, thereby improving the efficiency of compiler autotuning.

6.3.3 Integrating Coarse Offline and Fine-Grained Online Learning

While developing an accurate offline model for compiler autotuning is challenging, a coarse offline model is feasible. For instance, loop vectorisation generally improves performance in most cases. By combining dynamic features obtained from profiling the program at a specific optimisation level (e.g., -O3), such as performance counters, we can develop an initial offline model. This offline model can provide a preliminary exploration, while the online model can perform more fine-grained searches.

Future work could focus on integrating these coarse offline models with fine-grained online learning approaches. The offline model can guide the initial search space, reducing the number of evaluations needed for the fine-grained online autotuning. This hybrid approach could enhance the efficiency and effectiveness of the compiler autotuning process by leveraging the strengths of both offline and online models.

Deep reinforcement learning (DRL) could serve as a promising approach to realise this integration, as it is well-suited for sequential decision-making and can naturally combine coarse guidance from an offline model with adaptive fine-grained exploration in the online phase. Similarly, code similarity based initialisation could be incorporated to provide the online search with a better starting point, effectively bridging the offline knowledge and online optimisation process.

References

[1] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[2] Richard M Stallman et al. Using the GNU compiler collection. *Free Software Foundation*, 4(02), 2003.

[3] Amir H Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. A survey on compiler autotuning using machine learning. *ACM Computing Surveys (CSUR)*, 51(5):1–42, 2018.

[4] Zheng Wang and Michael O'Boyle. Machine learning in compiler optimization. *Proceedings of the IEEE*, 106(11):1879–1901, 2018.

[5] Mircea Trofin, Yundi Qian, Eugene Brevdo, Zinan Lin, Krzysztof Choromanski, and David Li. Mlgo: a machine learning guided compiler optimizations framework. *arXiv preprint arXiv:2101.04808*, 2021.

[6] 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.

[7] David Eriksson, Michael Pearce, Jacob Gardner, Ryan D Turner, and Matthias Poloczek. Scalable global optimization via local bayesian optimization. *Advances in Neural Information Processing Systems*, 32, 2019.

[8] Xingchen Wan, Vu Nguyen, Huong Ha, Binxin Ru, Cong Lu, and Michael A Osborne. Think global and act local: Bayesian optimisation over high-dimensional categorical and mixed search spaces. In *International Conference on Machine Learning*, pages 10663–10674. PMLR, 2021.

[9] Natalie Maus, Haydn Jones, Juston Moore, Matt J Kusner, John Bradshaw, and Jacob Gardner. Local latent space bayesian optimization over structured inputs. *Advances in neural information processing systems*, 35:34505–34518, 2022.

[10] Christopher KI Williams and Carl Edward Rasmussen. *Gaussian processes for machine learning*, volume 2. MIT press Cambridge, MA, 2006.

[11] S Hochreiter. Long short-term memory. *Neural Computation MIT-Press*, 1997.

[12] A Vaswani. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017.

[13] Laurent Valentin Jospin, Hamid Laga, Farid Boussaid, Wray Buntine, and Mohammed Bennamoun. Hands-on bayesian neural networks-a tutorial for deep learning users. *IEEE Computational Intelligence Magazine*, 17(2):29–48, 2022.

[14] Lelac Almagor, Keith D Cooper, Alexander Grosul, Timothy J Harvey, Steven W Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. Finding effective compilation sequences. *ACM SIGPLAN Notices*, 39(7):231–239, 2004.

[15] Stefano Cereda, Gianluca Palermo, Paolo Cremonesi, and Stefano Doni. A collaborative filtering approach for the automatic tuning of compiler optimisations. In *The 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 15–25, 2020.

[16] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. Spec cpu2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, pages 41–42, 2018.

[17] 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, 2015.

[18] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *International conference on learning and intelligent optimization*, pages 507–523. Springer, 2011.

[19] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. *Advances in neural information processing systems*, 25, 2012.

[20] Jasper Snoek, Oren Rippel, Kevin Swersky, Ryan Kiros, Nadathur Satish, Narayanan Sundaram, Mostofa Patwary, Mr Prabhat, and Ryan Adams. Scalable bayesian optimization using deep neural networks. In *International conference on machine learning*, pages 2171–2180. PMLR, 2015.

[21] Jost Tobias Springenberg, Aaron Klein, Stefan Falkner, and Frank Hutter. Bayesian optimization with robust bayesian neural networks. *Advances in neural information processing systems*, 29, 2016.

[22] Jacob Gardner, Geoff Pleiss, Kilian Q Weinberger, David Bindel, and Andrew G Wilson. Gpytorch: Blackbox matrix-matrix gaussian process inference with gpu acceleration. *Advances in neural information processing systems*, 31, 2018.

[23] Christopher M Bishop and Nasser M Nasrabadi. *Pattern recognition and machine learning*, volume 4. Springer, 2006.

[24] Matthias Seeger. Gaussian processes for machine learning. *International journal of neural systems*, 14(02):69–106, 2004.

[25] Maximilian Balandat, Brian Karrer, Daniel Jiang, Samuel Daulton, Ben Letham, Andrew G Wilson, and Eytan Bakshy. Botorch: A framework for efficient monte-carlo bayesian optimization. *Advances in neural information processing systems*, 33:21524–21538, 2020.

[26] Niranjan Srinivas, Andreas Krause, Sham M Kakade, and Matthias Seeger. Gaussian process optimization in the bandit setting: No regret and experimental design. *arXiv preprint arXiv:0912.3995*, 2009.

[27] Donald R Jones, Matthias Schonlau, and William J Welch. Efficient global optimization of expensive black-box functions. *Journal of Global optimization*, 13:455–492, 1998.

[28] James Wilson, Frank Hutter, and Marc Deisenroth. Maximizing acquisition functions for bayesian optimization. *Advances in neural information processing systems*, 31, 2018.

[29] James T Wilson, Riccardo Moriconi, Frank Hutter, and Marc Peter Deisenroth. The reparameterization trick for acquisition functions. *arXiv preprint arXiv:1712.00424*, 2017.

[30] Danilo Jimenez Rezende, Shakir Mohamed, and Daan Wierstra. Stochastic backpropagation and variational inference in deep latent gaussian models. In *International conference on machine learning*, volume 2, page 2, 2014.

[31] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., USA, 1st edition, 1989.

[32] Melanie Mitchell. *An introduction to genetic algorithms*. MIT press, 1998.

[33] Nikolaus Hansen, Sibylle D Müller, and Petros Koumoutsakos. Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (cma-es). *Evolutionary computation*, 11(1):1–18, 2003.

[34] Nikolaus Hansen and Andreas Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary computation*, 9(2):159–195, 2001.

[35] H-G Beyer and Dirk V Arnold. Theory of evolution strategies: A tutorial. *Theoretical aspects of evolutionary computing*, pages 109–133, 2001.

[36] Stefan Droste, Thomas Jansen, and Ingo Wegener. On the analysis of the (1+ 1) evolutionary algorithm. *Theoretical Computer Science*, 276(1-2):51–81, 2002.

[37] Benjamin Doerr and Carola Doerr. Optimal parameter choices through self-adjustment: Applying the 1/5-th rule in discrete settings. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 1335–1342, 2015.

[38] Benjamin Doerr and Frank Neumann. *Theory of evolutionary computation: Recent developments in discrete optimization*. Springer Nature, 2019.

[39] Keith D Cooper, Philip J Schielke, and Devika Subramanian. Optimizing for reduced code space using genetic algorithms. In *Proceedings of the ACM SIGPLAN 1999 workshop on Languages, compilers, and tools for embedded systems*, pages 1–9, 1999.

[40] Toru Kisuki, Peter MW Knijnenburg, and Michael FP O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *Proceedings 2000 International Conference on Parallel Architectures and Compilation Techniques (Cat. No. PR00622)*, pages 237–246. IEEE, 2000.

[41] Keith D Cooper, Devika Subramanian, and Linda Torczon. Adaptive optimizing compilers for the 21st century. *The Journal of Supercomputing*, 23:7–22, 2002.

[42] Spyridon Triantafyllis, Manish Vachharajani, Neil Vachharajani, and David I August. Compiler optimization-space exploration. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*, pages 204–215. IEEE, 2003.

[43] Prasad Kulkarni, Wankang Zhao, Hwashin Moon, Kyunghwan Cho, David Whalley, Jack Davidson, Mark Bailey, Yunheung Paek, and Kyle Gallivan. Finding effective optimization phase sequences. *ACM SIGPLAN Notices*, 38(7):12–23, 2003.

[44] Prasad Kulkarni, Stephen Hines, Jason Hiser, David Whalley, Jack Davidson, and Douglas Jones. Fast searches for effective optimization phase sequences. *ACM SIGPLAN Notices*, 39(6):171–182, 2004.

[45] Keith D Cooper, Alexander Grosul, Timothy J Harvey, Steven Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. Acme: adaptive compilation made efficient. *ACM SIGPLAN Notices*, 40(7):69–77, 2005.

[46] Prasad A Kulkarni, David B Whalley, Gary S Tyson, and Jack W Davidson. Exhaustive optimization phase order space exploration. In *International Symposium on Code Generation and Optimization (CGO'06)*, pages 13–pp. IEEE, 2006.

[47] Jack W. Davidson, Gary S. Tyson, David B. Whalley, and Prasad A. Kulkarni. Evaluating heuristic optimization phase order search algorithms. In *International Symposium on Code Generation and Optimization (CGO'07)*, pages 157–169, 2007.

[48] Kenneth Hoste and Lieven Eeckhout. Cole: compiler optimization level exploration. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, pages 165–174, 2008.

[49] Prasad A Kulkarni, David B Whalley, Gary S Tyson, and Jack W Davidson. Practical exhaustive optimization phase order exploration and evaluation. *ACM Transactions on Architecture and Code Optimization (TACO)*, 6(1):1–36, 2009.

[50] Thayalan Sandran, Nordin Zakaria, and Anindya Jyoti Pal. An optimized tuning of genetic algorithm parameters in compiler flag selection based on compilation and execution duration. In *Proceedings of the International Conference on Soft Computing for Problem Solving (SocProS 2011) December 20-22, 2011: Volume 2*, pages 599–610. Springer, 2012.

[51] Herbert Jordan, Peter Thoman, Juan J Durillo, Simone Pellegrini, Philipp Gschwandtner, Thomas Fahringer, and Hans Moritsch. A multi-objective auto-tuning framework for parallel codes. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2012.

[52] Yang Chen, Shuangde Fang, Yuanjie Huang, Lieven Eeckhout, Grigori Fursin, Olivier Temam, and Chengyong Wu. Deconstructing iterative optimization. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(3):1–30, 2012.

[53] Michael R Jantz and Prasad A Kulkarni. Exploiting phase inter-dependencies for faster iterative compiler optimization phase order searches. In *2013 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pages 1–10. IEEE, 2013.

[54] Suresh Purini and Lakshya Jain. Finding good optimization sequences covering program space. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):1–23, 2013.

[55] Cedric Nugteren and Valeriu Codreanu. Cltune: A generic auto-tuner for opencl kernels. In *2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip*, pages 195–202. IEEE, 2015.

[56] Luiz GA Martins, Ricardo Nobre, Joao MP Cardoso, Alexandre CB Delbem, and Eduardo Marques. Clustering-based selection for the exploration of compiler optimization sequences. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(1):1–28, 2016.

[57] Unai Garciarena and Roberto Santana. Evolutionary optimization of compiler flag selection by learning and exploiting flags interactions. In *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion*, pages 1159–1166, 2016.

[58] Ricardo Nobre, Luiz GA Martins, and João MP Cardoso. A graph-based iterative compiler pass selection and phase ordering approach. *ACM SIGPLAN Notices*, 51(5):21–30, 2016.

[59] Tao Wang, Nikhil Jain, David Beckingsale, David Boehme, Frank Mueller, and Todd Gamblin. Funcytuner: Auto-tuning scientific applications with per-loop compilation. In *Proceedings of the 48th International Conference on Parallel Processing*, ICPP '19, New York, NY, USA, 2019.

[60] Michael Kruse, Hal Finkel, and Xingfu Wu. Autotuning Search Space for Loop Transformations . In *2020 IEEE/ACM 6th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC) and Workshop on Hierarchical Parallelism for Exascale Computing (HiPar)*, Los Alamitos, CA, USA, November 2020.

[61] Junjie Chen, Ningxin Xu, Peiqi Chen, and Hongyu Zhang. Efficient compiler autotuning via bayesian optimization. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1198–1209. IEEE, 2021.

[62] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 303–316, 2014.

[63] Ari Rasch, Richard Schulze, Michel Steuwer, and Sergei Gorlatch. Efficient autotuning of parallel programs with interdependent tuning parameters via autotuning framework (atf). *ACM Transactions on Architecture and Code Optimization (TACO)*, 18(1):1–26, 2021.

[64] Jaehoon Koo, Prasanna Balaprakash, Michael Kruse, Xingfu Wu, Paul Hovland, and Mary Hall. Customized monte carlo tree search for llvm/polly's composable

loop optimization transformations. In *2021 International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 82–93. IEEE, 2021.

[65] Huanting Wang, Zhanyong Tang, Cheng Zhang, Jiaqi Zhao, Chris Cummins, Hugh Leather, and Zheng Wang. Automating reinforcement learning architecture design for code optimization. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*, pages 129–143, 2022.

[66] Sunghyun Park, Salar Latifi, Yongjun Park, Armand Behroozi, Byungsoo Jeon, and Scott Mahlke. Srtuner: Effective compiler optimization customization by exposing synergistic relations. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 118–130. IEEE, 2022.

[67] Chris Cummins, Bram Wasti, Jiadong Guo, Brandon Cui, Jason Ansel, Sahir Gomez, Somya Jain, Jia Liu, Olivier Teytaud, Benoit Steiner, et al. Compilergym: Robust, performant compiler optimization environments for ai research. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 92–105. IEEE, 2022.

[68] Mingxuan Zhu and Dan Hao. Compiler auto-tuning via critical flag selection. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1000–1011. IEEE, 2023.

[69] Mingxuan Zhu, Dan Hao, and Junjie Chen. Compiler autotuning through multiple-phase learning. *ACM Transactions on Software Engineering and Methodology*, 33(4):1–38, 2024.

[70] Mark Stephenson and Saman Amarasinghe. Predicting unroll factors using supervised classification. In *International symposium on code generation and optimization*, pages 123–134. IEEE, 2005.

[71] Felix Agakov, Edwin Bonilla, John Cavazos, Björn Franke, Grigori Fursin, Michael FP O'Boyle, John Thomson, Marc Toussaint, and Christopher KI Williams. Using machine learning to focus iterative optimization. In *International Symposium on Code Generation and Optimization (CGO'06)*, pages 11–pp. IEEE, 2006.

[72] John Cavazos, Grigori Fursin, Felix Agakov, Edwin Bonilla, Michael FP O'Boyle, and Olivier Temam. Rapidly selecting good compiler optimizations using performance counters. In *International Symposium on Code Generation and Optimization (CGO'07)*, pages 185–197. IEEE, 2007.

[73] Kapil Vaswani, Matthew J Thazhuthaveetil, YN Srikant, and PJ Joseph. Microarchitecture sensitive empirical models for compiler optimizations. In *International Symposium on Code Generation and Optimization (CGO'07)*, pages 131–143. IEEE, 2007.

[74] Tomofumi Yuki, Lakshminarayanan Renganarayanan, Sanjay Rajopadhye, Charles Anderson, Alexandre E Eichenberger, and Kevin O'Brien. Automatic creation of tile size selection models. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 190–199, 2010.

[75] Grigori Fursin, Yuriy Kashnikov, Abdul Wahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Bilha Mendelson, Ayal Zaks, Eric Courtois, et al. Milepost gcc: Machine learning enabled self-tuning compiler. *International journal of parallel programming*, 39:296–327, 2011.

[76] Sameer Kulkarni and John Cavazos. Mitigating the compiler optimization phase-ordering problem using machine learning. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 147–162, 2012.

[77] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. Petabricks: A language and compiler for algorithmic choice. *ACM Sigplan Notices*, 44(6):38–49, 2009.

[78] Yufei Ding, Jason Ansel, Kalyan Veeramachaneni, Xipeng Shen, Una-May O'Reilly, and Saman Amarasinghe. Autotuning algorithmic choice for input sensitivity. *ACM SIGPLAN Notices*, 50(6):379–390, 2015.

[79] Amir Hossein Ashouri, Giovanni Mariani, Gianluca Palermo, Eunjung Park, John Cavazos, and Cristina Silvano. Cobayn: Compiler autotuning framework using bayesian networks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(2):1–25, 2016.

[80] Chris Cummins, Pavlos Petoumenos, Michel Steuwer, and Hugh Leather. Autotuning opencl workgroup size for stencil patterns. In *6th International Workshop on Adaptive Self-tuning Computing Systems 2016*, 2016.

[81] Amir H Ashouri, Andrea Bignoli, Gianluca Palermo, Cristina Silvano, Sameer Kulkarni, and John Cavazos. Micomp: Mitigating the compiler phase-ordering problem using optimization sub-sequences and machine learning. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(3):1–28, 2017.

[82] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. End-to-end deep learning of optimization heuristics. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 219–232. IEEE, 2017.

[83] Rahim Mammadli, Ali Jannesari, and Felix Wolf. Static neural compiler optimization via deep reinforcement learning. In *2020 IEEE/ACM 6th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC) and Workshop on Hierarchical Parallelism for Exascale Computing (HiPar)*, pages 1–11. IEEE, 2020.

[84] Ameer Haj-Ali, Nesreen Ahmed, Ted Willke, Sophia Shao, Krste Asanovic, and Ion Stoica. Neurovectorizer: End-to-end vectorization with deep reinforcement learning. In *Proceedings of the 2020 International Symposium on Code Generation and Optimization*, CGO 2020. ACM, 2020.

[85] Anderson Faustino Da Silva, Bruno Conde Kind, José Wesley de Souza Magalhães, Jerônimo Nunes Rocha, Breno Campos Ferreira Guimaraes, and Fernando Magno Quinão Pereira. Anghabench: A suite with one million compilable c benchmarks for code-size reduction. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 378–390. IEEE, 2021.

[86] Anderson Faustino da Silva, Bernardo NB De Lima, and Fernando Magno Quintão Pereira. Exploring the space of optimization sequences for code-size reduction: insights and tools. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, pages 47–58, 2021.

[87] Hongzhi Liu, Jie Luo, Ying Li, and Zhonghai Wu. Iterative compilation optimization based on metric learning and collaborative filtering. *ACM Transactions on Architecture and Code Optimization (TACO)*, 19(1):1–25, 2021.

[88] Raphael Mosaner, David Leopoldseder, Lukas Stadler, and Hanspeter Mössenböck. Using machine learning to predict the code size impact of duplication heuristics in a dynamic compiler. In *Proceedings of the 18th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, pages 127–135, 2021.

[89] Shalini Jain, Yashas Andaluri, S VenkataKeerthy, and Ramakrishna Upadrasta. Poset-rl: Phase ordering for optimizing size and execution time using reinforcement learning. In *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 121–131. IEEE, 2022.

[90] Youwei Liang, Kevin Stone, Ali Shameli, Chris Cummins, Mostafa Elhoushi, Jiadong Guo, Benoit Steiner, Xiaomeng Yang, Pengtao Xie, Hugh James Leather, et al. Learning compiler pass orders using coreset and normalized value prediction. In *International Conference on Machine Learning*, pages 20746–20762. PMLR, 2023.

[91] Tamim Burgstaller, Damian Garber, Viet-Man Le, and Alexander Felfernig. Optimization space learning: A lightweight, noniterative technique for compiler autotuning. In *Proceedings of the 28th ACM International Systems and Software Product Line Conference*, pages 36–46, 2024.

[92] Haolin Pan, Yuanyu Wei, Mingjie Xing, Yanjun Wu, and Chen Zhao. Towards efficient compiler auto-tuning: Leveraging synergistic search spaces. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*, pages 614–627, 2025.

[93] Chris Cummins, Volker Seeker, Dejan Grubisic, Mostafa Elhoushi, Youwei Liang, Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Kim Hazelwood, Gabriel Synnaeve, et al. Large language models for compiler optimization. *arXiv preprint arXiv:2309.07062*, 2023.

[94] Scott Kirkpatrick, C Daniel Gelatt Jr, and Mario P Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[95] Pauline Bennet, Carola Doerr, Antoine Moreau, Jeremy Rapin, Fabien Teytaud, and Olivier Teytaud. Nevergrad: black-box optimization platform. *ACM SIGEVOlution*, 14(1):8–15, 2021.

[96] Ziyu Wang, Masrour Zoghi, Frank Hutter, David Matheson, Nando De Freitas, et al. Bayesian optimization in high dimensions via random embeddings. In *IJCAI*, pages 1778–1784. Citeseer, 2013.

[97] Ziyu Wang, Frank Hutter, Masrour Zoghi, David Matheson, and Nando De Feitas. Bayesian optimization in a billion dimensions via random embeddings. *Journal of Artificial Intelligence Research*, 55:361–387, 2016.

[98] Hong Qian, Yi-Qi Hu, and Yang Yu. Derivative-free optimization of high-dimensional non-convex functions by sequential random embeddings. In *IJCAI*, pages 1946–1952, 2016.

[99] Xiaoyu Lu, Javier Gonzalez, Zhenwen Dai, and Neil D Lawrence. Structured variationally auto-encoded optimization. In *International conference on machine learning*, pages 3267–3275. PMLR, 2018.

[100] Amin Nayebi, Alexander Munteanu, and Matthias Poloczek. A framework for bayesian optimization in embedded subspaces. In *International Conference on Machine Learning*, pages 4752–4761. PMLR, 2019.

[101] Mickaël Binois, David Ginsbourger, and Olivier Roustant. On the choice of the low-dimensional domain for global optimization via random embeddings. *Journal of global optimization*, 76(1):69–90, 2020.

[102] Ben Letham, Roberto Calandra, Akshara Rai, and Eytan Bakshy. Re-examining linear embeddings for high-dimensional bayesian optimization. *Advances in neural information processing systems*, 33:1546–1558, 2020.

[103] Kirthevasan Kandasamy, Jeff Schneider, and Barnabás Póczos. High dimensional bayesian optimisation and bandits via additive models. In *International conference on machine learning*, pages 295–304. PMLR, 2015.

[104] Zi Wang, Chengtao Li, Stefanie Jegelka, and Pushmeet Kohli. Batched high-dimensional bayesian optimization via structural kernel learning. In *International Conference on Machine Learning*, pages 3656–3664. PMLR, 2017.

[105] Jacob Gardner, Chuan Guo, Kilian Weinberger, Roman Garnett, and Roger Grosse. Discovering and exploiting additive structure for bayesian optimization. In *Artificial Intelligence and Statistics*, pages 1311–1319. PMLR, 2017.

[106] Paul Rolland, Jonathan Scarlett, Ilija Bogunovic, and Volkan Cevher. High-dimensional bayesian optimization via additive models with overlapping groups. In *International conference on artificial intelligence and statistics*, pages 298–307. PMLR, 2018.

[107] Mojmir Mutny and Andreas Krause. Efficient high dimensional bayesian optimization with additivity and quadrature fourier features. *Advances in Neural Information Processing Systems*, 31, 2018.

[108] Zi Wang, Clement Gehring, Pushmeet Kohli, and Stefanie Jegelka. Batched large-scale bayesian optimization in high-dimensional spaces. In *International Conference on Artificial Intelligence and Statistics*, pages 745–754. PMLR, 2018.

[109] Cheng Li, Sunil Gupta, Santu Rana, Vu Nguyen, Svetha Venkatesh, and Alistair Shilton. High dimensional bayesian optimization using dropout. *arXiv preprint arXiv:1802.05400*, 2018.

[110] Johannes Kirschner, Mojmir Mutny, Nicole Hiller, Rasmus Ischebeck, and Andreas Krause. Adaptive and safe bayesian optimization in high dimensions via one-dimensional subspaces. In *International Conference on Machine Learning*, pages 3429–3438. PMLR, 2019.

[111] Riccardo Moriconi, KS Sesh Kumar, and Marc Peter Deisenroth. High-dimensional bayesian optimization with projections using quantile gaussian processes. *Optimization Letters*, 14:51–64, 2020.

[112] Jian Tan, Niv Nayman, and Mengchang Wang. Cobbo: Coordinate backoff bayesian optimization with two-stage kernels. *arXiv preprint arXiv:2101.05147*, 2021.

[113] Santu Rana, Cheng Li, Sunil Gupta, Vu Nguyen, and Svetha Venkatesh. High dimensional bayesian optimization with elastic gaussian process. In *International conference on machine learning*, pages 2883–2891. PMLR, 2017.

[114] ChangYong Oh, Efstratios Gavves, and Max Welling. Bock: Bayesian optimization with cylindrical kernels. In *International Conference on Machine Learning*, pages 3868–3877. PMLR, 2018.

[115] David Eriksson and Martin Jankowiak. High-dimensional bayesian optimization with sparse axis-aligned subspaces. In *Uncertainty in Artificial Intelligence*, pages 493–503. PMLR, 2021.

[116] Linnan Wang, Rodrigo Fonseca, and Yuandong Tian. Learning search space partition for black-box optimization using monte carlo tree search. *Advances in Neural Information Processing Systems*, 33:19511–19522, 2020.

[117] John P Cunningham, Krishna V Shenoy, and Maneesh Sahani. Fast gaussian process methods for point process intensity estimation. In *Proceedings of the 25th international conference on Machine learning*, pages 192–199, 2008.

[118] Michalis Titsias. Variational learning of inducing variables in sparse gaussian processes. In *Artificial intelligence and statistics*, pages 567–574. PMLR, 2009.

[119] James Hensman, Nicolo Fusi, and Neil D Lawrence. Gaussian processes for big data. *arXiv preprint arXiv:1309.6835*, 2013.

[120] Andrew Wilson and Hannes Nickisch. Kernel interpolation for scalable structured gaussian processes (kiss-gp). In *International conference on machine learning*, pages 1775–1784. PMLR, 2015.

[121] James Hensman, Alexander Matthews, and Zoubin Ghahramani. Scalable variational gaussian process classification. In *Artificial Intelligence and Statistics*, pages 351–360. PMLR, 2015.

[122] Andrew G Wilson, Zhiting Hu, Russ R Salakhutdinov, and Eric P Xing. Stochastic variational deep kernel learning. *Advances in neural information processing systems*, 29, 2016.

[123] Diederik P Kingma. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[124] Paul Tseng. Convergence of a block coordinate descent method for nondifferentiable minimization. *Journal of optimization theory and applications*, 109:475–494, 2001.

[125] Thomas Bartz-Beielstein, Christian WG Lasarczyk, and Mike Preuß. Sequential parameter optimization. In *2005 IEEE congress on evolutionary computation*, volume 1, pages 773–780. IEEE, 2005.

[126] Frank Hutter, Holger H Hoos, Kevin Leyton-Brown, and Kevin P Murphy. An experimental investigation of model-based parameter optimisation: Spo and beyond. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 271–278, 2009.

[127] Frank Hutter, Holger H Hoos, Kevin Leyton-Brown, and Kevin Murphy. Time-bounded sequential parameter optimization. In *International Conference on Learning and Intelligent Optimization*, pages 281–298. Springer, 2010.

[128] Tim Head, Manoj Kumar, Holger Nahrstaedt, Gilles Louppe, and Iaroslav Shcherbatyi. scikit-optimize/scikit-optimize, October 2021. https://doi.org/10.5281/zenodo.5565057.

[129] Ruben Martinez-Cantin. Bayesopt: A bayesian optimization library for nonlinear optimization, experimental design and bandits. *Journal of Machine Learning Research*, 15(115):3915–3919, 2014.

[130] Kirthevasan Kandasamy, Karun Raju Vysyaraju, Willie Neiswanger, Biswajit Paria, Christopher R Collins, Jeff Schneider, Barnabas Poczos, and Eric P Xing. Tuning hyperparameters without grad students: Scalable and robust bayesian optimisation with dragonfly. *Journal of Machine Learning Research*, 21(81):1–27, 2020.

[131] Alexander I Cowen-Rivers, Wenlong Lyu, Zhi Wang, Rasul Tutunov, Hao Jianye, Jun Wang, and Haitham Bou Ammar. Hebo: Heteroscedastic evolutionary bayesian optimisation. *arXiv e-prints*, pages arXiv–2012, 2020.

[132] Victor Picheny, Joel Berkeley, Henry B Moss, Hrvoje Stojic, Uri Granta, Sebastian W Ober, Artem Artemev, Khurram Ghani, Alexander Goodall, Andrei

Paleyes, et al. Trieste: Efficiently exploring the depths of black-box functions with tensorflow. *arXiv preprint arXiv:2302.08436*, 2023.

[133] Nicolas Knudde, Joachim van der Herten, Tom Dhaene, and Ivo Couckuyt. Gpflowopt: A bayesian optimization library using tensorflow. *arXiv preprint arXiv:1711.03845*, 2017.

[134] Artur Souza, Luigi Nardi, Leonardo B Oliveira, Kunle Olukotun, Marius Lindauer, and Frank Hutter. Bayesian optimization with a prior for the optimum. In *Machine Learning and Knowledge Discovery in Databases. Research Track: European Conference, ECML PKDD 2021, Bilbao, Spain, September 13–17, 2021, Proceedings, Part III 21*, pages 265–296. Springer, 2021.

[135] Natalie Maus, Haydn T Jones, Juston S Moore, Matt J Kusner, John Bradshaw, and Jacob R Gardner. Local latent space bayesian optimization over structured inputs. *arXiv preprint arXiv:2201.11872*, 2022.

[136] Natalie Maus, Kaiwen Wu, David Eriksson, and Jacob Gardner. Discovering many diverse solutions with bayesian optimization. In *International Conference on Artificial Intelligence and Statistics*, pages 1779–1798. PMLR, 2023.

[137] Sebastian Ament, Samuel Daulton, David Eriksson, Maximilian Balandat, and Eytan Bakshy. Unexpected improvements to expected improvement for bayesian optimization. *Advances in Neural Information Processing Systems*, 36:20577–20612, 2023.

[138] Seunghun Lee, Jaewon Chu, Sihyeon Kim, Juyeon Ko, and Hyunwoo J Kim. Advancing bayesian optimization via learning correlated latent space. *Advances in Neural Information Processing Systems*, 36, 2024.

[139] The GPyOpt authors. GPyOpt: A bayesian optimization framework in python. http://github.com/SheffieldML/GPyOpt, 2016.

[140] Xingfu Wu, Michael Kruse, Prasanna Balaprakash, Hal Finkel, Paul Hovland, Valerie Taylor, and Mary Hall. Autotuning polybench benchmarks with llvm clang/polly loop optimization pragmas using bayesian optimization. *Concurrency and Computation: Practice and Experience*, 34(20):e6683, 2022.

[141] Rohan Basu Roy, Tirthak Patel, Vijay Gadepally, and Devesh Tiwari. Bliss: auto-tuning complex applications using a pool of diverse lightweight learning models. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 1280–1295, 2021.

[142] Vivek Nair, Zhe Yu, Tim Menzies, Norbert Siegmund, and Sven Apel. Finding faster configurations using flash. *IEEE Transactions on Software Engineering*, 46(7):794–811, 2018.

[143] Erik Orm Hellsten, Artur Souza, Johannes Lenfers, Rubens Lacouture, Olivia Hsu, Adel Ejjeh, Fredrik Kjolstad, Michel Steuwer, Kunle Olukotun, and Luigi Nardi. Baco: A fast and portable bayesian compiler optimization framework. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, pages 19–42, 2023.

[144] Jiayu Zhao, Chunwei Xia, , and Zheng Wang. Leveraging compilation statistics for compiler phase ordering. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2025.

[145] Francesco Barchi, Gianvito Urgese, Enrico Macii, and Andrea Acquaviva. Code mapping in heterogeneous platforms using deep learning and llvm-ir. In *Proceedings of the 56th Annual Design Automation Conference 2019*, pages 1–6, 2019.

[146] Ameer Haj-Ali, Qijing Jenny Huang, John Xiang, William Moses, Krste Asanovic, John Wawrzynek, and Ion Stoica. Autophase: Juggling hls phase orderings in random forests with deep reinforcement learning. *Proceedings of Machine Learning and Systems*, 2:70–81, 2020.

[147] Alexander Brauckmann, Andrés Goens, Sebastian Ertel, and Jeronimo Castrillon. Compiler-based graph representations for deep learning models of code. In *Proceedings of the 29th International Conference on Compiler Construction*, pages 201–211, 2020.

[148] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. Neural code comprehension: A learnable representation of code semantics. *Advances in neural information processing systems*, 31, 2018.

[149] S VenkataKeerthy, Rohit Aggarwal, Shalini Jain, Maunendra Sankar Desarkar, Ramakrishna Upadrasta, and YN Srikant. IR2vec: LLVM IR based scalable program embeddings. *ACM Transactions on Architecture and Code Optimization (TACO)*, 17(4):1–27, 2020.

[150] Chris Cummins, Zacharias V Fisches, Tal Ben-Nun, Torsten Hoefler, Michael FP O'Boyle, and Hugh Leather. Programl: A graph-based program representation for data flow analysis and compiler optimizations. In *International Conference on Machine Learning*, pages 2244–2253. PMLR, 2021.

[151] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. *Advances in neural information processing systems*, 24, 2011.

[152] José Miguel Hernández-Lobato, James Requeima, Edward O Pyzer-Knapp, and Alán Aspuru-Guzik. Parallel and distributed thompson sampling for large-scale accelerated exploration of chemical space. In *International conference on machine learning*, pages 1470–1479. PMLR, 2017.

[153] Daniel J Lizotte, Tao Wang, Michael H Bowling, Dale Schuurmans, et al. Automatic gait optimization with gaussian process regression. In *IJCAI*, volume 7, pages 944–949, 2007.

[154] Ruben Martinez-Cantin, Nando De Freitas, Eric Brochu, José Castellanos, and Arnaud Doucet. A bayesian exploration-exploitation approach for optimal online sensing and planning with a visually guided mobile robot. *Autonomous Robots*, 27(2):93–103, 2009.

[155] Peter I Frazier. A tutorial on bayesian optimization. *arXiv preprint arXiv:1807.02811*, 2018.

[156] Jian Wu and Peter Frazier. The parallel knowledge gradient method for batch bayesian optimization. *Advances in neural information processing systems*, 29, 2016.

[157] Zi Wang and Stefanie Jegelka. Max-value entropy search for efficient bayesian optimization. In *International Conference on Machine Learning*, pages 3627–3635. PMLR, 2017.

[158] Henry B Moss, David S Leslie, Javier Gonzalez, and Paul Rayson. Gibbon: General-purpose information-based bayesian optimisation. *Journal of Machine Learning Research*, 22(235):1–49, 2021.

[159] Jasper Snoek, Kevin Swersky, Rich Zemel, and Ryan Adams. Input warping for bayesian optimization of non-stationary functions. In *International Conference on Machine Learning*, pages 1674–1682. PMLR, 2014.

[160] Tanweer Alam, Shamimul Qamar, Amit Dixit, and Mohamed Benaida. Genetic algorithm: Reviews, implementations, and applications. *International Journal of Engineering Pedagogy*, 2020.

[161] Marcin Molga and Czesław Smutnicki. Test functions for optimization needs. *Test functions for optimization needs*, 101:48, 2005.

[162] Grigori Fursin and Olivier Temam. Collective optimization: A practical collaborative approach. *ACM Transactions on Architecture and Code Optimization (TACO)*, 7(4):1–29, 2010.

[163] Nikolaus Hansen, yoshihikoueno, ARF1, Kento Nozawa, Luca Rolshoven, Matthew Chan, Youhei Akimoto, brieglhostis, and Dimo Brockhoff. pycma: r3.2.2, March 2022. https://doi.org/10.5281/zenodo.6370326.

[164] J. Blank and K. Deb. pymoo: Multi-objective optimization in python. *IEEE Access*, 8:89497–89509, 2020.

[165] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ international conference on intelligent robots and systems*, pages 5026–5033. IEEE, 2012.

[166] Chris Ying, Aaron Klein, Eric Christiansen, Esteban Real, Kevin Murphy, and Frank Hutter. Nas-bench-101: Towards reproducible neural architecture search. In *International conference on machine learning*, pages 7105–7114. PMLR, 2019.

[167] Kenan Šehić, Alexandre Gramfort, Joseph Salmon, and Luigi Nardi. Lassobench: A high-dimensional hyperparameter optimization benchmark suite for lasso. *arXiv preprint arXiv:2111.02790*, 2021.

[168] Horia Mania, Aurelia Guy, and Benjamin Recht. Simple random search provides a competitive approach to reinforcement learning. *arXiv preprint arXiv:1803.07055*, 2018.

[169] Ricardo Nobre, Luiz GA Martins, and Joao MP Cardoso. Use of previously acquired positioning of optimizations for phase ordering exploration. In *Proceedings of the 18th international workshop on software and compilers for embedded systems*, pages 58–67, 2015.

[170] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 283–294, 2011.