Leveraging Compilation Statistics for Compiler Phase Ordering

Anonymous author(s)

***Abstract*—Choosing the optimal order and combination of compiler optimization passes - known as *phase ordering* - can significantly enhance the performance of compiled binaries. However, existing approaches struggle to capture the subtle interaction between compiler passes and waste time on low-profitable pass sequences. We introduce CITROEN, a better approach for compiler phase ordering. CITROEN leverages pass-related compilation statistics to reject low-profitable compiler pass sequences to reduce the overhead of phase ordering search. It employs Bayesian optimization to navigate the search space, using compilation statistics instead of traditional tuning parameters to build an online cost model that provides both the performance prediction and the prediction uncertainty of compilation configurations. It dynamically allocates search iterations across source files to optimize search time in multi-file programs. We evaluate CITROEN by integrating it with the LLVM v17.0 compiler and applying it to benchmarks from cBench and SPEC CPU 2017. CITROEN outperforms existing auto-tuning methods, discovering high-performing configurations quicker with fewer search iterations.**

***Index Terms*—compiler optimization, phase ordering, Bayesian optimation, compilation statistics**

I. INTRODUCTION

Compilers play a key role in the performance and energy optimization of computer systems. Modern compilers like LLVM [1] and GCC [2] offer a rich set of optimization passes [3], where a pass implements some specific code analysis and transformation techniques like loop unrolling, instruction scheduling and register allocation. By default, compilers provide settings such as -O3 for performance optimization and -Oz for code size reduction, which apply a bundle of passes like loop unrolling and vectorization in a fixed order. However, studies have shown that the optimal choice and ordering of passes can vary greatly across programs [4]. Carefully selecting and ordering compiler passes - a problem known as *phase ordering* - can significantly improve the application performance [5]. Phase ordering is particularly useful for frequently executed programs, as even a small improvement in the running time can be beneficial in the long run.

A significant challenge in phase ordering is the vast optimization 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 [6], making them hard to find in such a large space.

Search-based auto-tuning is widely used for phase ordering [3], [5], [7]–[15]. Unlike predictive modeling [16]–[19], 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 auto-tuning 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 optimizing 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 maximize the overall performance gains within the available budget.

We present CITROEN[1], a better search-based auto-tuning 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-vectorize* pass reports how many loops have been vectorized. If we observe a strong positive correlation between the number of vectorized loops and improved performance, we can infer that loop vectorization is likely to benefit the input program. In such cases, if changing the compiler pass sequences leads to a reduction in the number of vectorized 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 auto-tuning by avoiding the profiling of pass sequences that offer no performance gain. This is achieved through a customized Bayesian optimization (BO) method [20], which builds an online probabilistic cost model (known as *the surrogate model* in the machine learning community) to evaluate compiler pass sequences. Our cost model takes as input a feature vector consisting of compilation

[1] Code and data available at: [URL redacted for double-blind review].

statistics and predicts both the runtime performance and the prediction uncertainty. The cost model is dynamically and constantly updated during the search process using new profiling data 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 optimization [21]–[24] 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 [22], enabling or disabling a compiler flag [21], loop tile sizes [23], and loop unroll factors [24], 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.

We evaluate CITROEN by applying it to optimize the phase ordering of the LLVM compiler. We test the resulting compilation system on the cBench [25] and SPEC CPU 2017 [26] benchmark suites 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.

This paper makes the following contributions:

- It proposes the first auto-tuning approach to utilize pass-related compilation statistics for compiler phase ordering;
- It presents an adaptive BO scheme to dynamically allocate the search budget across multiple source files within a single program;
- It develops an open-source framework for multi-module compiler phase ordering.

II. BACKGROUND AND MOTIVATION

A. Program Scope

As depicted in Figure 1, CITROEN searches for an optimal compiler pass sequence for a given optimization goal. In this work, we focus on performance optimization, aiming to minimize execution time; however, CITROEN can also be applied

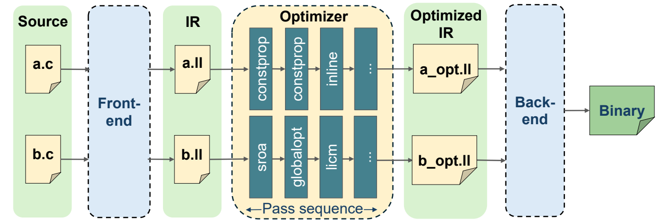

Figure 1: The CITROEN compiler flow for applying customized pass sequences.

```
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 2: 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.

to optimize other objectives, such as energy consumption. CITROEN supports programs with multiple source files (e.g., C programs with '.c' files), treating each file as an independent optimization unit, referred to as a *module*. Unlike previous approaches [3], [5], [7]–[13], [15], CITROEN allows different pass sequences for different modules, thereby expanding the search space and improving overall performance.

CITROEN works as follows: it first uses the compiler front-end (e.g., LLVM clang) to compile each module into unoptimized intermediate representations (IRs). Next, different compiler pass sequences are applied to these IRs to generate optimized versions. The optimized IRs are then compiled to assembly language by the llvm static compiler and then linked to final executable binary. In this work, a pass can be applied multiple times within a single pass sequence to optimize an individual source file, and our current implementation uses the default compiler parameters for each pass. In this work, we consider 76 LLVM transformation passes and a maximum compiler sequence of 120 passes (the transformation sequence length of '-O3' is 99).

B. Motivation

As a motivation example, consider applying phase ordering to LLVM v17.0 to optimize `telecom_gsm` from the cBench benchmark suite [25] 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 optimization here.

Table I: 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.NumVectorInstructions	**mem2reg.NumPHIInsert**	**mem2reg.NumPromoted**	**mem2reg.NumSingleStore**	**instcombine.NumCombined**	
1	*mem2reg slp-vectorizer*	**14**	21	43	29	0	**1.13×**
2	*slp-vectorizer mem2reg*	0	21	43	29	0	0.85×
3	*inst-combine mem2reg slp-vectorizer*	0	18	41	29	271	0.85×
4	*mem2reg inst-combine slp-vectorizer*	0	21	43	29	244	0.86×
5	*mem2reg slp-vectorizer instcombine*	**14**	21	43	29	164	**1.14×**

Figure 2a shows a hot code snippet that computes the dot product of two vectors, which would benefit from superword-level parallelism (SLP) vectorization. Sequentially applying the *mem2reg* and *slp-vectorizer* passes results in successful vectorization, as shown in Figure 2b. However, when the *inst-combine* pass is applied between *mem2reg* and *slp-vectorizer* (i.e., in the order of *'mem2reg, instcombine, slp-vectorizer'*), vectorization fails due to the profitability analysis. This is because *instcombine* optimizes greedily without considering the later vectorization opportunity. Specifically, as can be seen from Figure 2c, *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 vectorized horizontal reduction. Consequently, the LLVM vectorizer skips vectorization 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. We observe that pass-related compilation statistics can help us to capture the relationship between pass sequences and the performance.

Table I list LLVM compilation statistics for five different pass sequences, along with their runtime performance, using the -O3 optimisation level as a baseline. These statistics can be gathered using the '`-stats -stats-json`' flags of LLVM '`opt`' tool. Assume that the execution times for the first three pass sequences have already been obtained through profiling. 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 $1.13\times$ 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 parallelizing 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 maximize the overall performance. CITROEN is designed to address these challenges using BO as a search technique, as described in the next subsection.

C. Bayesian Optimization

Our rationale for using BO is that it offers a principled method for balancing *exploration* and *exploitation* [20]. In our context, exploitation refers to profiling pass sequences predicted to yield improved performance. In contrast, exploration focuses on profiling pass sequences, for which compilation statistics have been explored less. Balancing exploration and exploitation is crucial because the online cost model is not always accurate.

BO balances exploration and exploitation by measuring the uncertainty of the model predictions. We follow the common practice of BO using a Gaussian process (GP) [27] to build our cost model to predict the potential speedup of a pass sequence. The GP model not only estimates the performance gain but also quantifies the uncertainty of its estimation. An acquisition function is then used to evaluate the trade-off between exploration and exploitation. Commonly used acquisition functions include Expected Improvement (EI) [28] and Upper Confidence Bound (UCB) [29]. However, these acquisition functions are designed for standard BO, which directly models the relationship between input parameters (pass sequences) and output. CITROEN instead converts the compilation statistics into a numerical feature vector to be used by the cost model (Sec. III-C), necessitating modifications to the standard acquisition functions (Sec. III-D).

III. OUR APPROACH

A. Overview

Figure 3 depicts the workflow of CITROEN. At the core of CITROEN is a BO search component based on compilation statistics (Sec. III-B), which will interact with a user-defined task function (Sec. III-F) that defines how to compile and

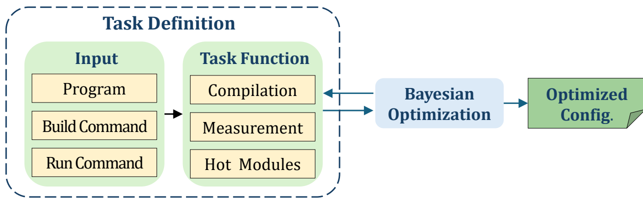

Figure 3: Overview of the CITROEN framework.

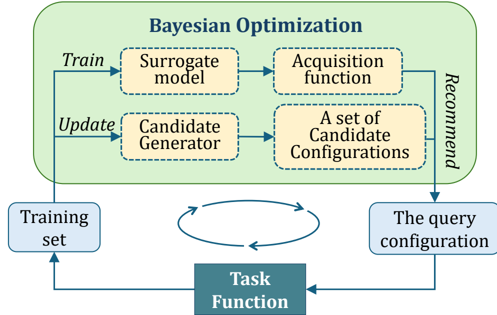

Figure 4: CITROEN's Bayesian optimization workflow.

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" optimization flag. Specifically, we use `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.

B. Bayesian Optimization for Compiler Tuning

Figure 4 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. III-C), an acquisition function for navigating the non-uniform, sparse feature space (Sec. III-D), and a GA-based pass sequence generator (Sec. III-E). Instead of running separate BO processes for each source file, CITROEN fits a global cost model to estimate the impact of individual module changes on overall program performance, dynamically determining which module to optimize 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, for m modules, CITROEN generates q candidate pass sequences per module (while fixing the pass sequence for other modules), resulting in $m * q$ candidate configurations. Finally, the acquisition function selects the highest-value configuration to

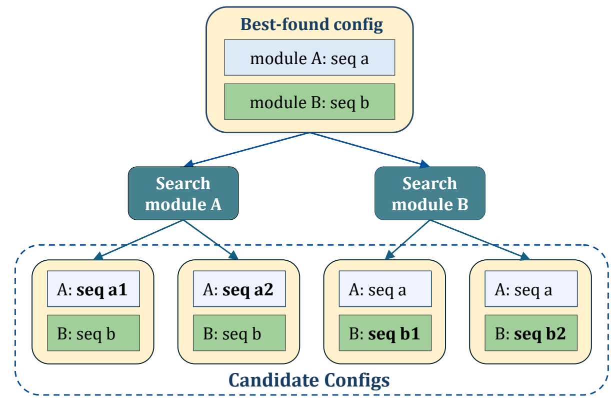

Figure 5: CITROEN's candidate configuration generator.

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 optimize next, allowing dynamic switching to maximize performance gains.

C. 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 [21], [24] 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.

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.

1) 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 customize the pass sequence for a given module. After excluding statistics unrelated to performance optimization (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 normalize 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, feature vectors are concatenated to represent the entire program.

2) 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 [24], [27]. The kernel function (describes the similarity between two inputs)

Table II: 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)	$2\sim 35$	$1\sim 22$	$1\sim 19$

of this model is defined by

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

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

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 minimizing the negative log marginal likelihood loss function [30]. 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.

3) Inference: Given a pass configuration c, we obtain its feature vector $\mathbf{x} = \varphi(c)$ by applying the pass sequence to the input program, collect the compilation statistics, from which we normalize the statistics values to obtain the 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

$$\mu(\mathbf{x}) = K(\mathbf{X},\mathbf{x})^T K(\mathbf{X},\mathbf{X})^{-1}\boldsymbol{y} \quad (3)$$

$$\sigma^2(\mathbf{x}) = k(\mathbf{x},\mathbf{x}) - K(\mathbf{X},\mathbf{x})^T K(\mathbf{X},\mathbf{X})^{-1} K(\mathbf{X},\mathbf{x}) \quad (4)$$

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.

D. Acquisition Function Design

Coverage issue. CITROEN fits the cost model in a non-uniform, sparse feature space where many statistic categories contain zero values for a given pass sequence. This 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 II, the training set fails to cover all statistic 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 prioritize 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) [28] as the acquisition function, defined by

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

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 on configurations with high model uncertainty, the model uncertainty itself is inaccurate for configurations with unexplored features because it does not consider the affect of unexplored features. As we will show in Sec. V-B, using standard EI for CITROEN could lead to suboptimal performance.

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

$$\alpha(\mathbf{x}) = \begin{cases} \text{EI}(\mathbf{x}) + 10^8, & \text{OOD} \\ \text{EI}(\mathbf{x}), & \text{not OOD} \end{cases} \quad (5)$$

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 unexplored features, we add a large number (10^8) to the EI function for OOD candidate configurations to make them have higher acquisition function values, thus being prioritized for selection.

E. Pass Sequence Generator

Due to the large number of possible pass sequences, it is impossible to evaluate the acquisition function values of all sequences. Like previous high-dimensional BO [31] CITROEN employs a GA sampling strategy to generate candidate samples in a large search space. Specifically, CITROEN maintains several top-performing pass sequences for each module as the GA population and applies mutation and crossover operations to this population to generate candidate offspring sequences.

Mutation. CITROEN's mutation involves randomly replacing a certain percentage of passes in a parent pass sequence in the population. Specifically, it applies random replacements to 10%, 20%, 50%, and 100% of the passes in the sequence, with each proportion having an equal probability.

Crossover. We combine two parent pass sequences in the population 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.

```
1  import citroen
2  from citroen.function_wrap import Function_wrap
3  from citroen.utils import gen_hotfiles
4  from citroen.BO.BO import BO
5  from fabric import Connection
6
7  # Define the task function
8  fun = Function_wrap(
9      # Explicitly declare the compiler as clangopt
10     build_cmd='make CC=clangopt',
11     build_dir='Example',
12     # User-defined run and evaluation command
13     run_and_eval_cmd='./run_eval.sh',
14     binary_name='a.out',
15     remote_run_dir='home/usr/RemoteExample',
16     ssh_connection=Connection(host="xxx.xxx.xxx") )
17
18 # Automatically recognize hotfiles
19 hotfiles = gen_hotfiles(fun)
20 fun.hotfiles = hotfiles
21
22 # Autotuning the phase order of the program
23 optimizer = BO(fun=fun, budget=1000)
24 best_cfg, best_cost = optimizer.minimize()
```

Figure 6: An example of using CITROEN for phase ordering.

F. Auto-tuning 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 [3], [14], [24] require users to manually re-implement the compilation process for different pass sequences, which requires engineering efforts, especially for programs with multiple source files. CITROEN simplifies this by automating the task function definition without manual re-implementation. As shown in Figure 6, 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 customized 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.

IV. EXPERIMENTAL SETUP

We implemented CITROEN in around 5K lines of Python and C/C++ code. It uses the GPyTorch [30] GP library to implement the GP regression process for the cost model.

In our experiments, the number of initial training samples (n_init) for the cost model and the number of candidate pass sequences (q) per iteration for each module is set to 20 and 500, respectively. At the beginning of the search, all candidate pass sequences are generated using the GA sampling strategy outlined in Sec. III-E. After using 1/4 of the total search iterations (number of measurements), CITROEN generates only 50 new candidate sequences per module, while the remaining $q - 50$ sequences are randomly selected from previously generated but unevaluated sequences. This approach ensures that the additional compilation overhead remains negligible compared to the overhead of program execution.

Table III: Benchmarks used in evaluation.

Suite	ID	Benchmark	#hot modules
cBench [25] (budget: 100/300/1000, platform: arm/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 [26] (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

Table IV: LLVM optimization 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-vectorize, 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

A. Benchmarks

Table III lists the benchmarks used in the experiments, including 26 programs in cBench [25] and 16 programs in SPEC CPU 2017 [26]. We only consider C/C++ programs that can be successfully compiled by LLVM v17.

B. Evaluation 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 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.

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

C. Competing Baselines

To evaluate the performance of CITROEN, we compare it with 5 prior autotuning approaches.

Random. While simple, random search is reported to be effective in previous work [9], [32], [33].

OpenTuner. This compiler auto-tuning framework [14] implements an ensemble of multiple evolutionary algorithms and can dynamically adjust its use of different algorithms.

Nevergrad. This search library [34] 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 [3] phase-ordering environment.

BOCA. This closely related work uses BO for the compiler flag selection problem [21]. It uses the random forest as its cost (surrogate) model. When applying it to the phase-ordering problem, the approach is adapted to use one-hot encoding to handle the input sequence used by the random forest model.

BaCO. This is a BO framework for compilation optimization [24]. It can handle different parameter types and thus can be directly used for the compiler phase-ordering problem.

To apply these baselines to optimize 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 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-autotuning a module, we will initialize the search algorithm using the best-found sample from the search history. In this way, these baselines will not waste too much time on modules that have limited potential for performance improvement.

D. Evaluation Methodology

We aim to answer the following research questions:

RQ1: How does CITROEN compare with prior autotuning approaches (Sec. V-A)?
RQ2: How do individual components of CITROEN contribute to its overall performance (Sec. V-B)?
RQ3: How do CITROEN's pass-related compilation statistics compare with existing feature extraction methods (Sec. V-C)?

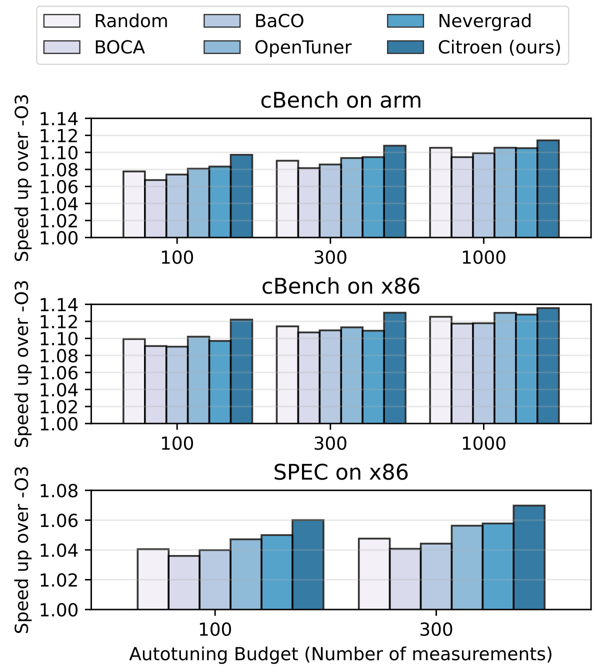

Figure 7: Average (geometric) performance on cBench and SPEC CPU 2017 with different search iteration budgets.

Performance report. Following [35], [36], we set three different budgets of 100, 300 and 1000 iterations for cBench and two different budgets of 100 and 300 iterations for SPEC CPU 2017 benchmark suite. The maximum budget of SPEC CPU 2017 benchmarks is 300 because of the program's long execution time. For each search iteration, we execute the compiled binary multiple times until the relative standard error of the mean execution time is below 1% (it typically requires 3-20 executions for cBench and 3 executions for SPEC), then report the mean execution time as feedback to the search algorithm. When reporting the performance after the search, we re-execute the best-found binary multiple times until the relative standard error of the mean is below 0.3% to obtain a more accurate result. For each search method, we report the average performance by repeating the tuning process five times for each benchmark.

V. EXPERIMENTAL RESULTS

A. Comparison with Baselines

Figure 7 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 $1.096\times$ speedup over -O3 compared to other methods' $1.067\times-1.083\times$ speedup. With a moderate budget of 300 iterations, CITROEN attains a $1.11\times$ speedup, which other methods require 1000 iterations to match.

To see how CITROEN generalizes across different benchmarks, we show in Figure 8 the comparison of CITROEN

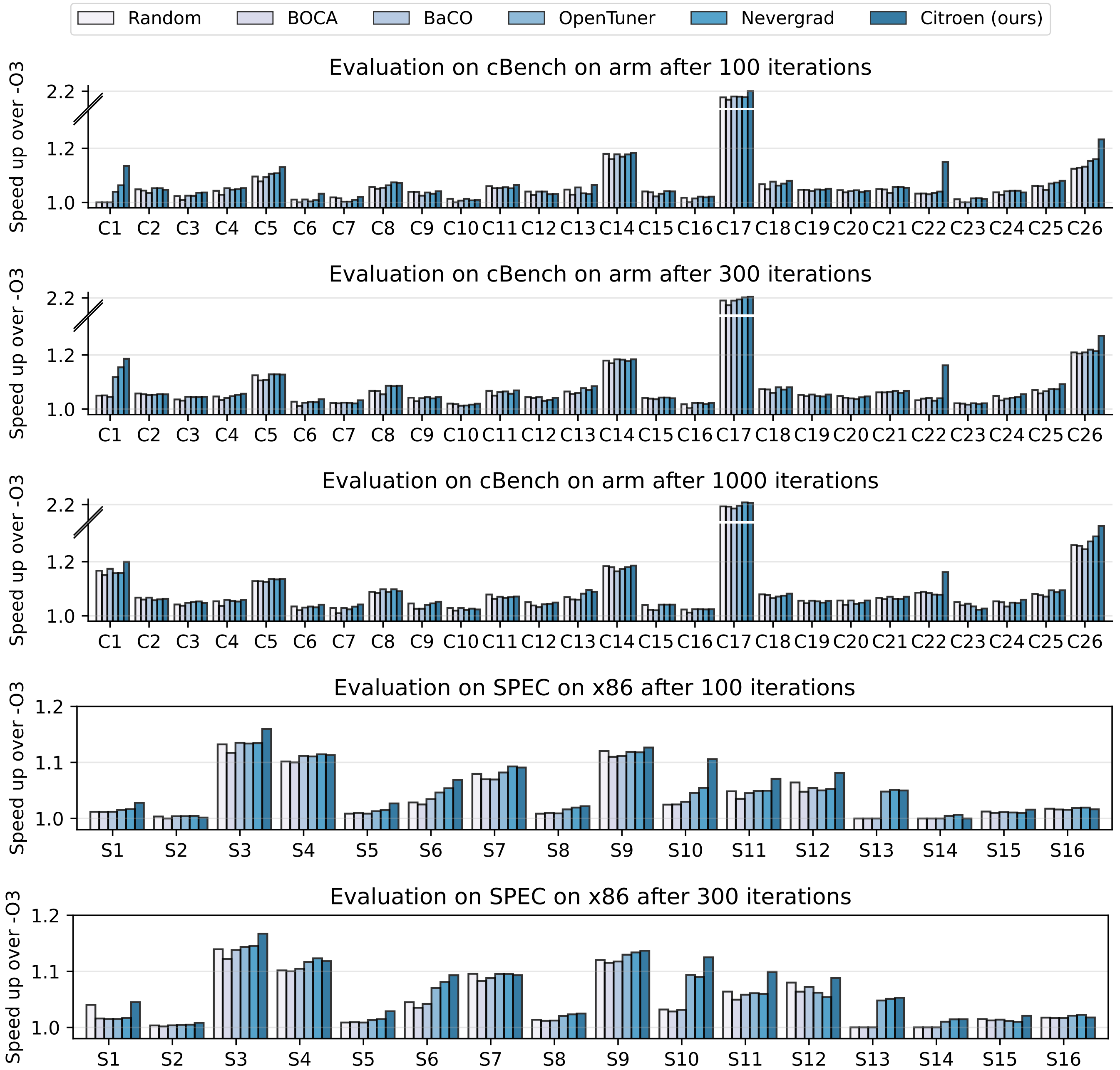

Figure 8: Evaluation on cBench and SPEC with different search iteration budgets.

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 $1.1\times$ speedup, one of its hot modules `bitcnts` should be optimized 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. 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 [9], [32], [33], where random search is reported to be effective enough in many cases.

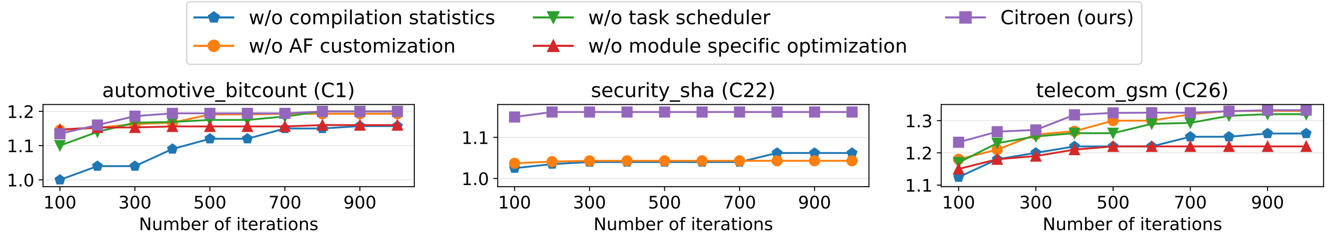

Figure 9: Ablation study on different benchmarks. The y-axis is the speedup relative to -O3. `security_sha` only owns one hot module, thus "task scheduler" and "module specific optimization" are not applicable to such single-module cases.

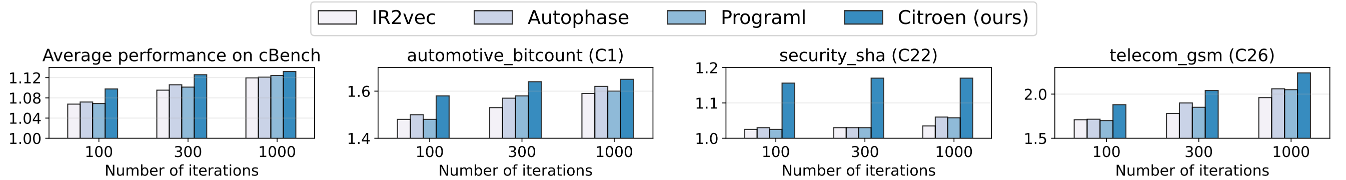

Figure 10: Comparision of CITROEN 's compilation statistics with alternative feature extraction methods using LLVM 10 as the compiler. The y-axis is the speedup relative to -O3.

B. 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 AF customization" employs standard EI as the acquisition function, ignoring coverage. "W/o task scheduler" sequentially auto-tunes each module instead of using a global task scheduler. "W/o module-specific optimization" applies a single pass sequence for all modules.

As can be seen in Figure 9, "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 AF customization" uses 1000 search iterations to achieve only $1.04\times$ speedup in `security_sha` (C22) while CITROEN uses 100 iterations to achieve $1.16\times$ speedup, showing that the coverage issue could significantly harm performance performance in some cases. For programs with multiple hot modules, "W/o module specific optimization" performs the worst in terms of the final achieved performance, revealing the effectiveness of module-specific optimization. As depicted in "W/o task scheduler", one-by-one autotuning could achieve module-specific optimization 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.

C. Comparision with alternative feature extraction methods

While our work uses pass-related compilation statistics to help autotuning, some works extract features from the intermediate representations (IRs) to build offline supervised/reinforcement learning models to directly predict the optimal compilation configuration. IR2vec [37], Autophase [38] and Programl [39] are three representative feature extraction methods. IR2vec combines representation learning methods with flow information to represent IRs as distributed embeddings in continuous space. Autophase develops analysis passes to extract static features from the IRs. Programl uses a graph-based program representation to accurately capture the semantics of programs and employs inst2vec [40] to obtain continuous embedding vectors. Although these works are not designed for pure search-based autotuning, their feature extraction methods could be used in our approach to build an online cost model and thus should be compared.

Figure 10 shows how our compilation statistics perform on the arm platform compared to alternative feature extraction methods, including IR2vec [37], Autophase [38], and Programl [39]. 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 recognized by IR2vec, Autophase and Programl, as *function-attrs* only changes the function attributes which are not considered in those feature extraction methods.

Table V 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 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

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

`bitcnts` in `automotive_bitcount`		`sha` in `security_sha`		`long_term` in `telecom_gsm`	
compilation statistics	performance change if disabling related passes	compilation statistics	performance change if disabling related passes	compilation statistics	performance change if disabling related passes
loop-unroll.NumUnrolled	-49%	*instcombine.NumCombined*	-27%	*mem2reg.NumPHIInsert*	-31%
inline.NumInlined	-43%	*mem2reg.NumPHIInsert*	-21%	*SLP.NumVectorInstructions*	-19%
licm.NumHoisted	-54%	*loop-rotate.NumRotated*	-26%	*instcombine.NumCombined*	-5%
mem2reg.NumPHIInsert	-52%	*early-cse.NumCSE*	-16%	*loop-vectorize.LoopsVectorized*	-7%
loop-unswitch.NumBranches	-22%	*loop-unroll.NumUnrolled*	-5%	*simplifycfg.NumSimpl*	-5%

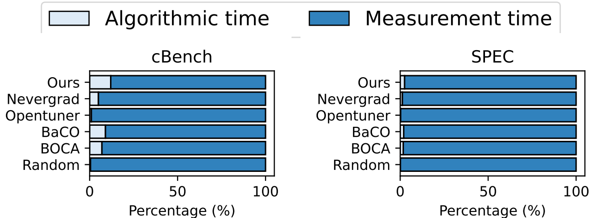

Figure 11: Average proportion of algorithmic runtime.

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 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.

D. Algorithmic Runtime

In Figure 11, 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 optimizing larger programs like SPEC CPU 2017.

VI. RELATED WORK

A. Compiler Phase Ordering

An extensive body of work shows compiler phase ordering can improve application performance [4], [41]. Prior works of compiler phase ordering often take an evolutionary search approach like genetic algorithms or simulated annealing [5], [7]–[13]. OpenTuner [14] and Nevergrad [34] are two representative search-based frameworks that have been used for compiler phase ordering [3]. Furthermore, random search is reported to be effective, as well as more sophisticated algorithms for exploring the optimisation space in many works [9], [32], [33]. While promising, prior works usually apply a single pass sequence to multiple source files. Our work takes a different approach by allowing different compiler pass sequences for individual files, leading to larger search spaces. By utilizing pass-related compilation statistics, our approach allows more efficient autotuning in the larger search spaces.

There are attempts to build a predictive model to directly predict the compiler phase order using supervised/reinforcement learning [16]–[19], [38]. As collecting sufficient training samples to cover the high-dimensional phase ordering optimization space is difficult, prior approaches reduce the search space by grouping compiler phases into sub-sequences. Our approach can be used to explore the search space to generate training samples for building a predictive model.

B. Bayesian Optimization for Program Autotuning

Some works have employed BO for program tuning. These include BOCA [21], Bliss [22], Ytopt [23], and BaCO [24]. BOCA uses the random forest as its surrogate model for the compiler flag selection problem. Bliss utilizes an ensemble of diverse Gaussian process models and acquisition functions to autotune parallel applications. Ytopt uses the traditional Skopt [42] BO library to optimize LLVM Clang/Polly pragma configurations on the PolyBench benchmark suite. BaCO customizes its BO implementation to support different parameter types and constraints for kernel optimization. While these frameworks show effectiveness in their tasks, they do not apply to the compiler phase-ordering problem. This is because they use the original tuning parameters as the input to fit their surrogate models. Unlike these prior works, CITROEN leverages the pass-related compilation statistics to design the surrogate model and the acquisition function. This improves performance when optimizing a complex search space introduced by compiler phase ordering.

VII. CONCLUSION

We have presented CITROEN, a BO-based search framework for compiler phase ordering. By leveraging pass-related compilation statistics to build an online probabilistic cost model, CITROEN avoids profiling pass sequences that offer no performance gain and implements a dynamical budget allocation across source files to support module-specific phase-ordering. Our evaluation shows that CITROEN outperforms existing approaches by achieving comparable tuning results using one-third of their search budget.

REFERENCES

[1] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE, 2004, pp. 75–86.

[2] R. M. Stallman *et al.*, “Using the gnu compiler collection,” *Free Software Foundation*, vol. 4, no. 02, 2003.

[3] C. Cummins, B. Wasti, J. Guo, B. Cui, J. Ansel, S. Gomez, S. Jain, J. Liu, O. Teytaud, B. Steiner *et al.*, “Compilergym: Robust, performant compiler optimization environments for ai research,” in *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2022, pp. 92–105.

[4] S. Cereda, G. Palermo, P. Cremonesi, and S. 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*, 2020, pp. 15–25.

[5] S. Purini and L. Jain, “Finding good optimization sequences covering program space,” *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 4, pp. 1–23, 2013.

[6] Z. Wang and M. O’Boyle, “Machine learning in compiler optimization,” *Proceedings of the IEEE*, vol. 106, no. 11, pp. 1879–1901, 2018.

[7] P. Kulkarni, W. Zhao, H. Moon, K. Cho, D. Whalley, J. Davidson, M. Bailey, Y. Paek, and K. Gallivan, “Finding effective optimization phase sequences,” *ACM SIGPLAN Notices*, vol. 38, no. 7, pp. 12–23, 2003.

[8] P. Kulkarni, S. Hines, J. Hiser, D. Whalley, J. Davidson, and D. Jones, “Fast searches for effective optimization phase sequences,” *ACM SIGPLAN Notices*, vol. 39, no. 6, pp. 171–182, 2004.

[9] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. O’Boyle, J. Thomson, M. Toussaint, and C. K. Williams, “Using machine learning to focus iterative optimization,” in *International Symposium on Code Generation and Optimization (CGO’06)*. IEEE, 2006, pp. 11–pp.

[10] P. A. Kulkarni, D. B. Whalley, G. S. Tyson, and J. W. Davidson, “Practical exhaustive optimization phase order exploration and evaluation,” *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 6, no. 1, pp. 1–36, 2009.

[11] R. Nobre, L. G. Martins, and J. M. 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*, 2015, pp. 58–67.

[12] L. G. Martins, R. Nobre, J. M. Cardoso, A. C. Delbem, and E. Marques, “Clustering-based selection for the exploration of compiler optimization sequences,” *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 13, no. 1, pp. 1–28, 2016.

[13] R. Nobre, L. G. Martins, and J. M. Cardoso, “A graph-based iterative compiler pass selection and phase ordering approach,” *ACM SIGPLAN Notices*, vol. 51, no. 5, pp. 21–30, 2016.

[14] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O’Reilly, and S. Amarasinghe, “Opentuner: An extensible framework for program autotuning,” in *Proceedings of the 23rd international conference on Parallel architectures and compilation*, 2014, pp. 303–316.

[15] A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano, “A survey on compiler autotuning using machine learning,” *ACM Computing Surveys (CSUR)*, vol. 51, no. 5, pp. 1–42, 2018.

[16] A. H. Ashouri, A. Bignoli, G. Palermo, C. Silvano, S. Kulkarni, and J. Cavazos, “Micomp: Mitigating the compiler phase-ordering problem using optimization sub-sequences and machine learning,” *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 3, pp. 1–28, 2017.

[17] H. Liu, J. Luo, Y. Li, and Z. Wu, “Iterative compilation optimization based on metric learning and collaborative filtering,” *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 19, no. 1, pp. 1–25, 2021.

[18] R. Mammadli, A. Jannesari, and F. 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)*. IEEE, 2020, pp. 1–11.

[19] S. Jain, Y. Andaluri, S. VenkataKeerthy, and R. 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)*. IEEE, 2022, pp. 121–131.

[20] P. I. Frazier, “A tutorial on bayesian optimization,” *arXiv preprint arXiv:1807.02811*, 2018.

[21] J. Chen, N. Xu, P. Chen, and H. Zhang, “Efficient compiler autotuning via bayesian optimization,” in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1198–1209.

[22] R. B. Roy, T. Patel, V. Gadepally, and D. 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*, 2021, pp. 1280–1295.

[23] X. Wu, M. Kruse, P. Balaprakash, H. Finkel, P. Hovland, V. Taylor, and M. Hall, “Autotuning polybench benchmarks with llvm clang/polly loop optimization pragmas using bayesian optimization,” *Concurrency and Computation: Practice and Experience*, vol. 34, no. 20, p. e6683, 2022.

[24] E. O. Hellsten, A. Souza, J. Lenfers, R. Lacouture, O. Hsu, A. Ejjeh, F. Kjolstad, M. Steuwer, K. Olukotun, and L. 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*, 2023, pp. 19–42.

[25] G. Fursin and O. Temam, “Collective optimization: A practical collaborative approach,” *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 7, no. 4, pp. 1–29, 2010.

[26] J. Bucek, K.-D. Lange, and J. v. Kistowski, “Spec cpu2017: Next-generation compute benchmark,” in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, 2018, pp. 41–42.

[27] C. K. Williams and C. E. Rasmussen, *Gaussian processes for machine learning*. MIT press Cambridge, MA, 2006, vol. 2, no. 3.

[28] J. Snoek, H. Larochelle, and R. P. Adams, “Practical bayesian optimization of machine learning algorithms,” *Advances in neural information processing systems*, vol. 25, 2012.

[29] N. Srinivas, A. Krause, S. M. Kakade, and M. Seeger, “Gaussian process optimization in the bandit setting: No regret and experimental design,” *arXiv preprint arXiv:0912.3995*, 2009.

[30] J. Gardner, G. Pleiss, K. Q. Weinberger, D. Bindel, and A. G. Wilson, “Gpytorch: Blackbox matrix-matrix gaussian process inference with gpu acceleration,” *Advances in neural information processing systems*, vol. 31, 2018.

[31] J. Zhao, R. Yang, S. QIU, and Z. Wang, “Unleashing the potential of acquisition functions in high-dimensional bayesian optimization,” *Transactions on Machine Learning Research*.

[32] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. F. O’Boyle, and O. Temam, “Rapidly selecting good compiler optimizations using performance counters,” in *International Symposium on Code Generation and Optimization (CGO’07)*. IEEE, 2007, pp. 185–197.

[33] Y. Chen, S. Fang, Y. Huang, L. Eeckhout, G. Fursin, O. Temam, and C. Wu, “Deconstructing iterative optimization,” *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 3, pp. 1–30, 2012.

[34] P. Bennet, C. Doerr, A. Moreau, J. Rapin, F. Teytaud, and O. Teytaud, “Nevergrad: black-box optimization platform,” *ACM SIGEVOlution*, vol. 14, no. 1, pp. 8–15, 2021.

[35] G. Fursin, Y. Kashnikov, A. W. Memon, Z. Chamski, O. Temam, M. Namolaru, E. Yom-Tov, B. Mendelson, A. Zaks, E. Courtois *et al.*, “Milepost gcc: Machine learning enabled self-tuning compiler,” *International journal of parallel programming*, vol. 39, pp. 296–327, 2011.

[36] S. Park, S. Latifi, Y. Park, A. Behroozi, B. Jeon, and S. Mahlke, “Srtuner: Effective compiler optimization customization by exposing synergistic relations,” in *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2022, pp. 118–130.

[37] S. VenkataKeerthy, R. Aggarwal, S. Jain, M. S. Desarkar, R. Upadrasta, and Y. Srikant, “Ir2vec: Llvm ir based scalable program embeddings,” *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 17, no. 4, pp. 1–27, 2020.

[38] A. Haj-Ali, Q. J. Huang, J. Xiang, W. Moses, K. Asanovic, J. Wawrzynek, and I. Stoica, “Autophase: Juggling hls phase orderings in random forests with deep reinforcement learning,” *Proceedings of Machine Learning and Systems*, vol. 2, pp. 70–81, 2020.

[39] C. Cummins, Z. V. Fisches, T. Ben-Nun, T. Hoefler, M. F. O’Boyle, and H. Leather, “Programl: A graph-based program representation for data flow analysis and compiler optimizations,” in *International Conference on Machine Learning*. PMLR, 2021, pp. 2244–2253.

[40] T. Ben-Nun, A. S. Jakobovits, and T. Hoefler, “Neural code comprehension: A learnable representation of code semantics,” *Advances in neural information processing systems*, vol. 31, 2018.

[41] L. Almagor, K. D. Cooper, A. Grosul, T. J. Harvey, S. W. Reeves, D. Subramanian, L. Torczon, and T. Waterman, “Finding effective

compilation sequences," *ACM SIGPLAN Notices*, vol. 39, no. 7, pp. 231–239, 2004.

[42] T. Head, M. Kumar, H. Nahrstaedt, G. Louppe, and I. Shcherbatyi, "scikit-optimize/scikit-optimize," Oct. 2021, https://doi.org/10.5281/zenodo.5565057.