Leveraging Compilation Statistics for Compiler Phase Ordering

Jiayu Zhao
University of Leeds
scjzh@leeds.ac.uk

Chunwei Xia
University of Leeds
c.xia@leeds.ac.uk

Zheng Wang
University of Leeds
z.wang5@leeds.ac.uk

***Abstract*—Choosing the optimal order and combination of compiler optimization passes - known as *phase ordering* - can 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 compiler and applying it to benchmarks from cBench and SPEC CPU 2017. CITROEN outperforms existing autotuning 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 autotuning is widely used for phase ordering [3], [5], [7]–[15]. Unlike predictive modeling [16]–[26], 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. Similarly, recognizing passes that degrade performance - by, for instance, blocking useful compiler optimization - helps prevent the algorithm from wasting time on measuring sequences that offer low-performance gain. Unfortunately, modeling the impact of a compiler pass is challenging, as its effect depends on the input program, its interactions with other passes, and their execution order. For example, loop unrolling can affect the efficiency 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 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-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.

[1]Code and data of this work are available at: https://github.com/gloaming2dawn/LLVMTuner

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 customized Bayesian optimization (BO) method [27], which builds an online probabilistic cost model (known as *the surrogate model*) to evaluate compiler pass sequences. Our cost model takes as input a feature vector consisting of compilation 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 uses an acquisition function to avoid profiling sequences likely to result in poor performance while prioritizing uncertain regions for exploration. For multi-module programs, it trains a global cost model by concatenating compilation statistics from individual source files, allowing dynamic allocation of the search budget to modules with the highest performance potential.

A key distinction between CITROEN and previous BO approaches in compiler optimization [28]–[31] 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 [29], enabling or disabling a compiler flag [28], loop tile sizes [30], and loop unroll factors [31], 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 [32] and SPEC CPU 2017 [33] benchmark suites on ARM and AMD x86 CPUs. Compared to state-of-the-art evolutionary and BO-based autotuning methods, CITROEN achieves similar results with only one-third of the 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 random search and up to 10% over the strongest baseline.

This paper makes the following two contributions:

- It introduces the first autotuning approach that leverages pass-related compilation statistics for compiler phase ordering (Sec. III);
- It proposes an adaptive BO scheme to dynamically allocate the search budget across multiple source files within a program (Sec. III-B).

II. BACKGROUND AND MOTIVATION

A. Program Scope

As depicted in Figure 1, CITROEN finds a compiler pass sequence for a given optimization goal. In this work, we focus on minimizing execution time; however, CITROEN can also be applied 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.

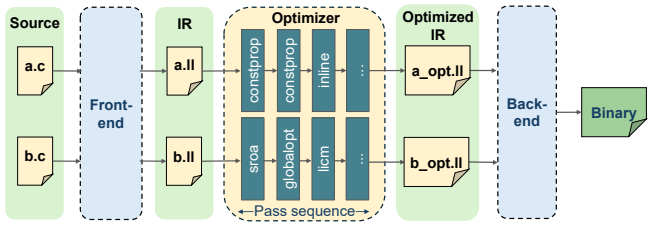

Figure 1: The CITROEN compiler flow for applying customized pass sequences.

```
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.

The CITROEN pipeline 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 code using the LLVM static compiler before being linked to the 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; by comparison, the transformation sequence length of '-O3' is 99.

B. Motivation

As a motivating example, consider phase ordering in LLVM (v17.0) to optimize `telecom_gsm` from the cBench suite [32] on an ARM Cortex-A57 CPU (Jetson TX2). In this benchmark, the `long_term` module (`long_term.c`) accounts for over 50% of execution time and is the target for optimization.

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 computing a dot product, which benefits from superword-level parallelism (SLP) vectorization. Applying *mem2reg* followed by *slp-vectorizer* enables successful vectorization (Figure 2b). However, inserting *instcombine* between them (i.e., *'mem2reg, instcombine, slp-vectorizer'*) prevents vectorization due to 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 lists the LLVM compilation statistics for five different pass sequences, along with their runtime performance, using the -O3 compiliation 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× 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.

How to correlate compilation statistics with performance to model the interactions of compiler passes? To this end, we find ways to map compilation statistics to performance and use the mapping model as a utility function to guide the search. Since this correlation depends on the input program, we iteratively refine the model during autotuning as more profiling data becomes available. For multi-module programs, effective budget allocation across modules is crucial for maximizing performance. CITROEN addresses these challenges using BO as a search technique, detailed in the next subsection.

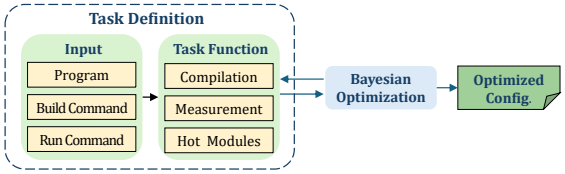

Figure 3: Overview of the CITROEN framework.

C. Bayesian Optimization

Our work leverages BO as it provides a principled approach to balancing *exploration* and *exploitation* [27]. In our context, exploitation profiles pass sequences expected to improve performance, while exploration prioritizes less-explored sequences based on compilation statistics. This balance is essential, as 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) [34] 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) [35] and Upper Confidence Bound (UCB) [36]. However, these acquisition functions are designed for standard BO, which directly models the relationship between input parameters (e.g., 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), requiring a customized acquisition function (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

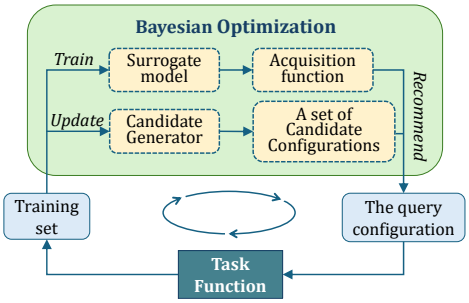

Figure 4: CITROEN's Bayesian optimization workflow.

task function (Sec. III-F) 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" compilation flag. During profiling, we 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 (or surrogate) 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 integrates 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. Our customized 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. It decides which module to optimize next for multiple hot modules, allowing dynamic switching to maximize performance gains.

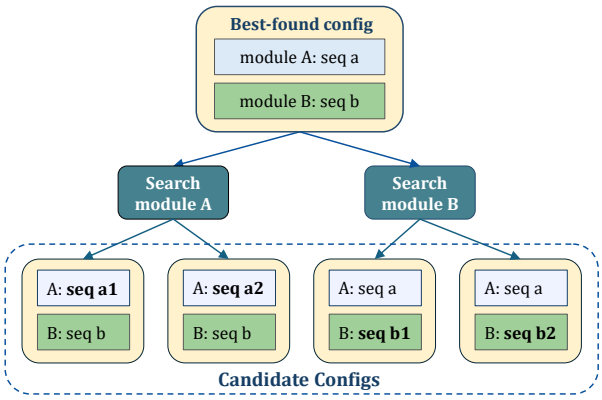

Figure 5: CITROEN's candidate configuration generator.

C. Surrogate Model for Performance Estimation

The cost (or surrogate) model of CITROEN is a utility function to approximate the optimization objective (i.e., speedup over -O3 in this work). Prior work in BO-based compiler tuning [28], [31] 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 represents compilation statistics as a numerical feature vector. These statistics are collected by enabling the `-stats -stats-json` flags in LLVM's `opt` tool when customizing pass sequences for a given module. After filtering out non-optimization-related statistics (e.g., analysis pass statistics), up to 255 statistics remain, though typically 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, where most values are zero due to inactive passes. For programs with multiple hot modules, feature vectors are concatenated to represent the entire program.

2) Model architecture and training: CITROEN use the Gaussian process with the Matérn-5/2 kernel to build the cost model because it is proven to be effective in prior BO applications [31], [34]. The kernel function (describes the similarity between two inputs) of this model is defined by:

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

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

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 [37]. Initially,

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)	$2 \sim 35$	$1 \sim 22$	$1 \sim 19$

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 and collecting normalized compilation statistics. Note that we do not execute the generated binary at this step; instead, we pass the feature vector to the GP to estimate the speedup (over $-O3$) with mean $\mu(\mathbf{x})$ and variance $\sigma^2(\mathbf{x})$:

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

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

where $\mathbf{X}$ and $\boldsymbol{y}$ are training inputs and speedup labels, respectively. The kernel matrix $K(\mathbf{X}, \mathbf{X})$ has entries $K(\mathbf{X}, \mathbf{X})_{i,j} = k(\mathbf{x}_i, \mathbf{x}_j)$, and $K(\mathbf{X}, \mathbf{x})$ contains kernel values between training points and the test point, $K(\mathbf{X}, \mathbf{x})_i = k(\mathbf{x}_i, \mathbf{x})$. The mean and variance are given to an acquisition function to select the next compilation configuration for profiling based on the predicted gain (i.e., speedup) and uncertainty (variance).

D. Acquisition Function Design

One of the challenges that CITROEN faces is to fit the cost model in a sparse, non-uniform feature space, where many statistic categories contain zeros for a given pass sequence, leading to coverage issues in the initial training set. Unlike standard BO, which benefits from uniform input space coverage, CITROEN cannot directly sample in the feature space. As a result, generated candidates may include unseen non-zero statistic categories. To illustrate this point, we applied 2,000 random pass sequences to cBench programs, selecting 20, 50, and 100 sequences per module for the initial training set for training the cost model. 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. To address this, the acquisition function should prioritize configurations with such features when determining exploration and exploitation.

Our acquisition function selects the next compilation configuration, aiming to balance the trade-off between exploiting high predicted values $\mu(\mathbf{x})$ and exploring regions with high uncertainty $\sigma(\mathbf{x})$. Standard BO often employs expected improvement (EI) [35] as the acquisition function:

$$\mathrm{EI}(\mathbf{x}) = \mathbb{E}\big[\max(f^* - y, 0) \mid y \sim \mathcal{N}(\mu(\mathbf{x}), \sigma^2(\mathbf{x}))\big] \tag{5}$$

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

However, standard BO acquisition functions do not adequately address the coverage issue in CITROEN. While EI encourages exploration in uncertain regions, it does not account for the presence of unexplored features, leading to unreliable uncertainty estimates. As shown in Sec. V-B, applying standard EI in CITROEN results in suboptimal performance. To mitigate this, we design a customized acquisition function $\alpha(\mathbf{x})$ for compiler phase ordering, based on EI:

$$\alpha(\mathbf{x}) = \begin{cases} \mathrm{EI}(\mathbf{x}) + 10^8, & \text{if OOD} \\ \mathrm{EI}(\mathbf{x}), & \text{otherwise} \end{cases} \tag{6}$$

where out-of-distribution (OOD) refers to candidate points $\mathbf{x}$ containing a non-zero feature (compilation statistic) unseen from the training set. To promote the exploration of configurations with unseen features, we add a large constant (10^8) to the EI function for OOD candidates, prioritizing their 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 [38] CITROEN employs a GA sampling strategy to generate candidate samples in a large search space. Specifically, CITROEN maintains 20 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. More sophisticated pass sequence generators are orthogonal to our work and can be easily integrated with our framework.

Mutation. Our mutation strategy randomly replaces 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 implement a one-point crossover by selecting a single crossover point in each parent pass sequence and swapping the segments beyond that point to generate new sequences.

F. Autotuning Task Definition

For phase order autotuning, users are traditionally required to define a task function that compiles a program with a given configuration and measures the performance of the resulting binary. Since compilers like LLVM do not support direct phase order specification per module, prior autotuning frameworks [3], [14], [31] require users to manually re-implement the compilation process for different pass sequences. This process incurs significant engineering effort, particularly for programs with multiple source files.

CITROEN automates this process, eliminating the need for manual re-implementation. As shown in Figure 6, CITROEN leverages the program's existing build script (e.g., a makefile) and provides a compiler driver (*clangopt* at line 10) to orchestrate compilation. In the example given in Figure 6, *clangopt* becomes the C compiler (i.e., `CC=clangopt`). It reads the compilation configuration from a JSON file before invoking the target compiler (e.g., `clang`) to compile the target file. Running the build script with *clangopt* automates

```
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
19 # Automatically recognize hotfiles
20 hotfiles = gen_hotfiles(fun)
21 fun.hotfiles = hotfiles
22
23 # Autotuning the phase order of the program
24 optimizer = BO(fun=fun, budget=1000)
25 best_cfg, best_cost = optimizer.minimize()
```

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

Table III: 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

multiple compilation commands, each using the specified configurations. Additionally, CITROEN supports automatic hot module detection (line 19) and remote execution (line 16), further reducing the effort required for phase ordering.

IV. EXPERIMENTAL SETUP

A. Implementation

We implemented CITROEN in around 5K lines of Python code. We use the GPyTorch [37] GP library to implement the GP regression process as the cost model of BO.

B. 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 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.

Table IV: Benchmarks used in evaluation.

Suite	ID	Benchmark	#hot modules
cBench [32] (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 [33] (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

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

C. Benchmarks

Table IV lists the benchmarks used in the experiments, including 26 programs from cBench [32] and 16 programs from SPEC CPU 2017 [33]. We only consider C/C++ programs that can be successfully compiled by LLVM v17.

D. 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 [9], [39], [40].

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 [41] 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 *compiler flag selection* [28]. 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 [31]. It can handle different parameter types and thus can be directly used for the compiler phase-ordering problem.

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. V-C, we compare CITROEN against three feature extraction methods: IR2vec [42], Autophase [43], and Programl [44].

E. Evaluation Methodology

Hyper-parameters of CITROEN. 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 GA sampling strategy described in Sec. III-E. 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. IV-D) 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 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 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 source files and will have little room for performance improvement.

Performance report. Following [45], [46], 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.

V. EXPERIMENTAL RESULTS

Our evaluation tries to answer the following 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)?

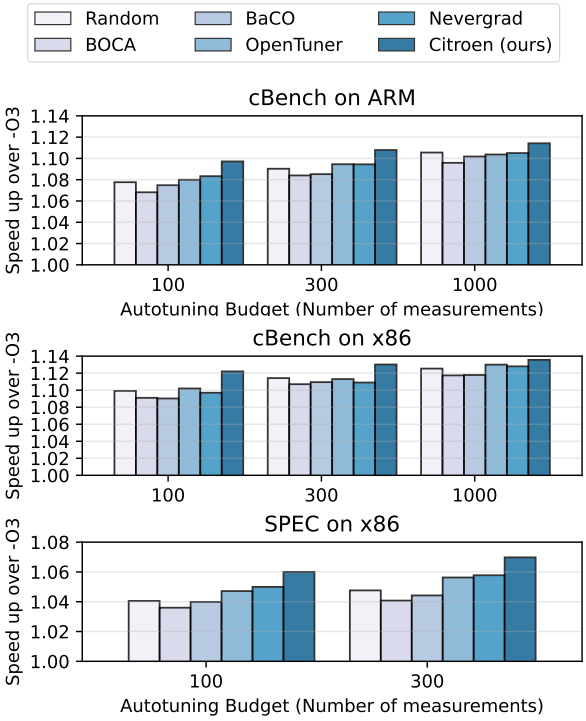

Figure 7: Geometric mean performance on cBench and SPEC CPU 2017 with different search iteration budgets.

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 evaluate how CITROEN generalizes across different benchmarks, Figure 8 compares its performance against baselines on individual benchmarks. CITROEN achieves significant improvements on several benchmarks, such as C1, C22, and C26. These benchmarks benefit from specific transformations, but the required compilation sequences are sparse in the search space. For instance, in C1 (`automotive_bitcount`), achieving more than $1.1\times$ speedup requires optimizing the hot module `bitcnts` using three loop transformations: *loop-unswitch*, *loop-unroll*, and *licm*. However, all baselines struggle to identify a pass sequence activating all three within a budget of 100 profiling measurements. Furthermore, for C22, we discovered the combination of *early-cse*, *instcombine*, *loop-rotate*, and *loop-fusion* passes to successfully unlock loop-level optimization. For S10, the key is to apply *loop-unroll* before and after the *instcombine* pass to enhance the instruction level parallelism. For S11, we improve vectorization by identifying a sub-sequence that applies *slp-vectorizer*

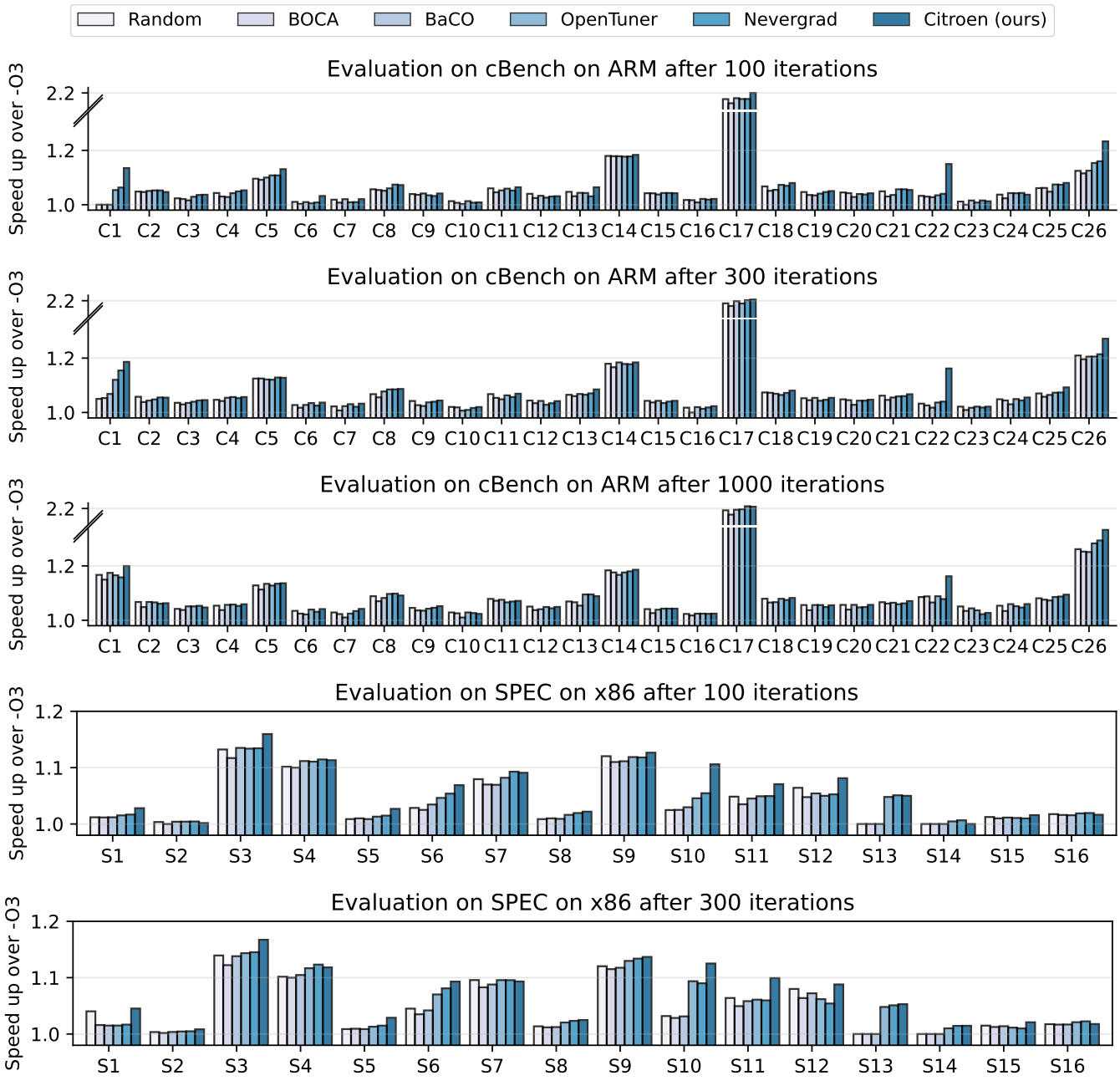

Figure 8: Evaluation on cBench and SPEC with different search iteration budgets.

after *sroa* and *simplifycfg*.

Another key observation is that for many benchmarks, all methods achieve similar performance. These benchmarks exhibit performance convergence under a small search budget (100) and a larger budget (1000) due to dominance by easily activated optimizations, such as *mem2reg*. This observation is consistent with previous studies [9], [39], [40], which report that random search is often sufficient in such cases.

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 shown in Figure 9, without utilizing compilation statistics, "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

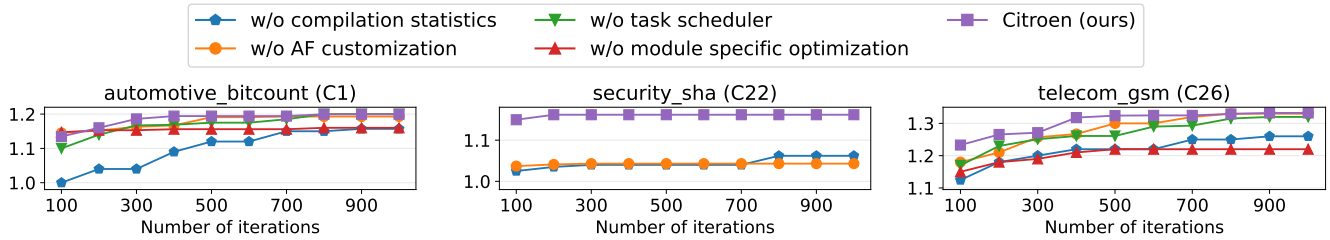

Figure 9: Ablation study on different benchmarks. The y-axis is the speedup relative to -O3. `security_sha` only owns one hot module, thus "task scheduler" and "module specific optimization" are not applicable to such single-module cases.

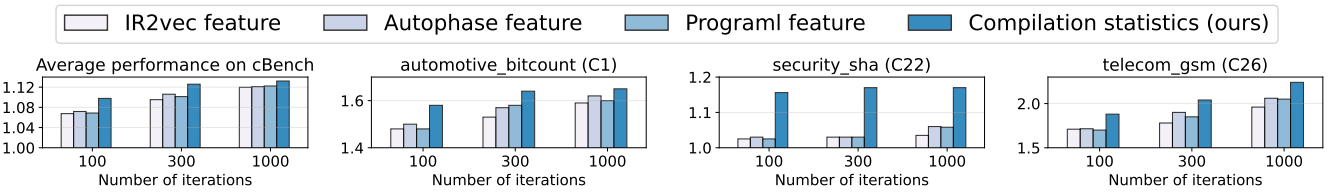

Figure 10: Impact of replacing compilation statistics with alternative feature extraction methods in CITROEN (using LLVM 10 as the compiler). The y-axis shows the speedup relative to -O3.

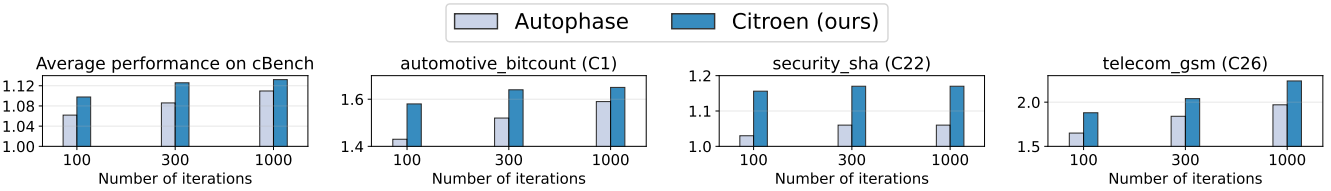

Figure 11: Comparison of CITROEN and Autophase using LLVM 10 as the compiler. The y-axis is the speedup relative to -O3.

statistics are a key component of CITROEN. "W/o AF customization" uses 1,000 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. Alternative Feature Extraction Methods

Prior works in machine learning-based compiler optimization 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 [42], Autophase [43], and Programl [44] provides three representative 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 [47] 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 10 evaluates our compilation statistic-based feature extraction approach against alternative IR2vec, Autophase, and Programl, on the Jetson TX2 ARM platform. Since Autophase and Programl support only LLVM 10, we use LLVM 10 in this experiment for a fair comparison. CITROEN, leveraging pass-related compilation statistics, clearly outperforms these methods. This is because alternative feature extractors struggle to distinguish transformations introduced by different passes. For example, the *function-attrs* pass can significantly impact performance for programs like `automotive_bitcount`, but IR2vec, Autophase, and Programl fail to capture its effects, as *function-attrs* only change function attributes, which these methods do not consider.

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 optimization (PPO) model on 100 randomly generated programs from Csmith [48], following the approach in Autophase. Then, we

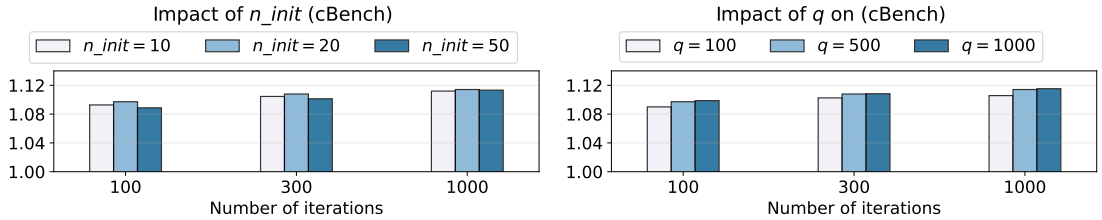

Figure 12: Hyperparameter Sensitivity Analysis of CITROEN. The y-axis is the speedup relative to -O3.

Table V: Top 5 impactful compilation statistics recognized by the CITROEN cost model on selected CBench benchmarks. Performance degradation is measured after removing the relevant passes from the final pass sequence and comparing the resulting performance against -O3 on the ARM platform.

`bitcnts` in `automotive_bitcount`		`sha` in `security_sha`		`long_term` in `telecom_gsm`	
compilation statistics	performance degradation without related passes	compilation statistics	performance degradation without related passes	compilation statistics	performance degradation without 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%

used this model as the initial policy for further RL-based search. Figure 11 reports the results, where CITROEN still consistently outperforms Autophase.

D. 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 refhyper reports how different hyperparameter values affect CITROEN's average performance across cBench benchmarks on the ARM platform. CITROEN demonstrates overall robustness to different hyperparameter values, except 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.

E. Compilation Statistics Analysis

Table V attempts to quantify the relationship between compilation statistics and performance speedup. For each program, we analyze the module (source file) with the longest runtime, running CITROEN for 1000 iterations to determine 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 influential compilation statistics, where a smaller lengthscale signifies a greater impact on performance. To assess each feature's importance, we measure the performance change after removing passes associated with that feature from the final pass sequence. For instance, if *loop-unroll.NumUnrolled* is identified as impactful, we remove *loop-unroll* from the sequence and observe the performance effect. The results show that impactful statistics vary across programs, indicating different optimization sensitivities. However, certain statistics consistently emerge as influential, highlighting their importance and relevance in compiler optimization.

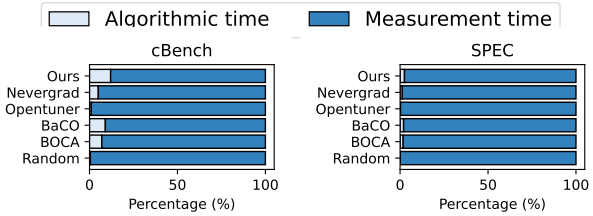

Figure 13: Average proportion of algorithmic runtime.

Table VI: Average search time (per benchmark) after 1000 search iterations in cBench and 300 iterations in SPEC.

	cBench (min)	SPEC (hours)
Ours	95	31.2
Nevergrad	88	30.6
Opentuner	82	30.0
BaCO	92	30.8
BOCA	90	30.8
Random	85	33.2

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 some passes are naturally correlated and often appear together in sequences to make the occurrence of certain statistics possible. For example, *loop-vectorize.LoopsVectorized* usually at least requires the sequential application of both the *loop-rotate* and *loop-vectorize* 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.

F. Algorithmic Runtime

Figure 13 presents the average proportion of algorithmic runtime (excluding objective function evaluation) across dif-

Table VII: Impact of program size on compilation and profiling overhead.

Benchmark	Hot File Line Count	Total Line Count	Hot File Compilation Time (s)	Profiling Time (s)
C2 (automotive_qsort1)	189	416	0.2	1
C1 (automotive_bitcount)	282	2288	0.2	1
C26 (telecom_gsm)	1575	23185	0.6	1
S7 (519.lbm_r)	723	930	1.0	120
S5 (510.parest_r)	5436	427000	2.0	190
S6 (511.povray)	13938	170000	5.0	240

ferent methods over 1000 search iterations in cBench and 300 iterations in SPEC CPU 2017. Since CITROEN requires additional parallel compilation (only for hot modules) and model training/inference, its algorithmic runtime is higher than that of other methods. However, this overhead remains negligible compared to the total performance measurement time, which includes program compilation and execution, particularly for larger programs like those in SPEC CPU 2017. Specially, the overhead of collecting statistics is small, accounting for less than 0.05% of the compilation overhead.

Table VI reports the raw wall-clock times of each search algorithm, where the overhead of CITROEN under the same number of search iterations is on par with the baselines. Notably, the additional search time of CITROEN is easily amortized, as it finds a binary with performance comparable to the best-performing baseline while requiring significantly fewer profiling runs - often just one-third of the total profiling times needed by other methods.

We also provide results showing how the change in program size affects the compilation and profiling overhead when using -O3 as the optimization level, as shown in Table VII. As the program size increases, the compilation time tends to increase due to the larger number of instructions and more complex dependencies that need to be resolved. However, the program size does not show a clear correlation with profiling time, as the profiling time is influenced by various other factors, such as algorithm complexity and input data size.

VI. Related Work

A. Compiler Phase Ordering

An extensive body of work shows compiler phase ordering can improve application performance [4], [49]. 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 [41] 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], [39], [40]. 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 predict the compiler phase order using supervised or reinforcement learning [18], [19], [22], [24], [43]. 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 [28], Bliss [29], Ytopt [30], and BaCO [31]. BOCA uses the random forest as its surrogate model for turning on or off compiler flags - a problem that usually has a smaller search space than the phase ordering problem targeted in this work. Bliss utilizes an ensemble of diverse Gaussian process models and acquisition functions to tune parallel applications. Ytopt uses Skopt [50], a standard Python BO library, to optimize LLVM Clang/Polly pragma configurations. BaCO customizes its BO implementation to support different parameter types and constraints for kernel optimization. While these frameworks show effectiveness in their tasks, they are not optimized for compiler phase ordering. 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 of 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. Future work could incorporate prior knowledge of pass correlations into our framework to further accelerate the search process, making it even more efficient.

Acknowledgments

This work was supported in part by the UK Engineering and Physical Sciences Research Council (EPSRC) under grant agreements EP/X018202/1 and EP/X037304/1. For any correspondence regarding this work, please contact Chunwei Xia (Email: c.xia@leeds.ac.uk) and Zheng Wang (Email: z.wang5@leeds.ac.uk).

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] Z. Wang and M. F. O'Boyle, "Mapping parallelism to multi-cores: a machine learning based approach," in *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2009, pp. 75–84.

[17] D. Grewe, Z. Wang, and M. F. O'Boyle, "Portable mapping of data parallel programs to opencl for heterogeneous systems," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2013, pp. 1–10.

[18] 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.

[19] 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.

[20] Z. Wang, G. Tournavitis, B. Franke, and M. F. O'boyle, "Integrating profile-driven parallelism detection and machine-learning-based mapping," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 11, no. 1, pp. 1–26, 2014.

[21] Z. Wang and M. F. O'Boyle, "Partitioning streaming parallelism for multi-cores: a machine learning based approach," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, 2010, pp. 307–318.

[22] 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.

[23] Z. Wang and M. F. O'boyle, "Using machine learning to partition streaming programs," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 10, no. 3, pp. 1–25, 2013.

[24] 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.

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

[26] P. Zhang, J. Fang, T. Tang, C. Yang, and Z. Wang, "Auto-tuning streamed applications on intel xeon phi," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2018, pp. 515–525.

[27] P. I. Frazier, "A tutorial on bayesian optimization," *arXiv preprint arXiv:1807.02811*, 2018.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

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

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

[36] 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.

[37] 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.

[38] 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*.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, 2011, pp. 283–294.

[49] 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.

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