WarmTuner: Program-Specific Warm Starts for Compiler Autotuning via Offline-to-Online Reinforcement Learning

1st Tianlu Qiao
School of Electronic and Computer Engineering
Peking University Shenzhen Graduate School
Shenzhen, China
tlqiao25@stu.pku.edu.cn

2nd Mingxuan Zhu
Key Lab of High Confidence Software Technologies (Peking University), Ministry of Education
Peking University
Beijing, China
zhumingxuan@stu.pku.edu.cn

3rd Zeyu Sun
National Key Laboratory of Space Integrated Information System
Institute of Software, Chinese Academy of Sciences
Beijing, China
zeyu.zys@gmail.com

4th Dan Hao*
School of Electronic and Computer Engineering
Peking University Shenzhen Graduate School
Shenzhen, China
haodan@pku.edu.cn

***Abstract*—Compilers are fundamental software tools that translate high-level programs into machine code. Modern compilers expose hundreds of optimizations, each turned on or off through an optimization flag, to improve the performance of the generated code. However, the number of possible flag combinations grows exponentially, making it difficult to find a flag configuration well suited to a given target program. Existing compiler auto-tuning techniques reduce tuning cost by pruning the search space, injecting search biases, or predicting configuration performance. Although some exploit program features, the knowledge they extract from historical data is frozen once search begins; runtime feedback then guides only the search itself, never the prior. As a result, when this prior mismatches the target program, these methods waste much of the limited online budget before the search reaches good configurations. We propose WarmTuner, an offline-to-online reinforcement learning framework that instead turns historical records into a program-conditioned *policy* that predicts each flag's setting over the full flag space and remains adaptable on the target program. Offline, WarmTuner learns this program-conditioned policy over the full flag space from historical good configurations. Online, it refines the same policy on the target program using real compile-run feedback, so that the policy is driven by measured speedups rather than limited to the historical data. Because each reward is expensive, yet directly verifiable, we instantiate the online update with Group Relative Policy Optimization (GRPO), which derives its signal from the relative speedups of candidates in the same round and does not need a separate value model—a natural fit for compiler tuning. To evaluate the performance of WarmTuner, we conducted an extensive experimental study using the latest version of the GCC compiler 15.2.0 on two widely used benchmarks, cBench and PolyBench. The results show that WarmTuner achieves an average speedup of $1.732\times$ over GCC -O3 and obtains the best result on 14/30 programs, significantly outperforming the compared techniques.**

***Index Terms*—Compiler, Compiler Auto-tuning, Reinforcement Learning, Group Relative Policy Optimization**

*Dan Hao is the corresponding author.

I. Introduction

Compilers are a core part of modern software development, which transform programs written in high-level languages into executable machine code. During this process, compiler can apply compiler optimizations to achieve objectives such as reducing the size of the compiled code or improving its runtime performance. Although GCC provides default optimization settings such as `-O1`, `-O2`, and `-O3`, these settings are designed for general use and do not always perform well on a specific program, possibly resulting in suboptimal runtime performance. It is therefore valuable to find a program-specific optimization setting, particularly for programs that run frequently or whose execution is time-consuming. This problem is commonly studied as compiler auto-tuning, which includes selecting which compiler optimizations to apply (i.e., *flag selection*) and deciding the order in which they are applied (i.e., *phase ordering*) [1]. As phase ordering does not apply to most production compilers such as GCC, this paper focuses on flag selection.

Common compilers like GCC provide hundreds of optimizations (e.g., loop unrolling), each controlled by enabling or disabling a specific optimization flag (e.g., `-funroll-loops`). Enabling an optimization flag does not always improve runtime performance, and its effect often depends on the other enabled flags and on the structure of the target program. Because of the large number of flags and their exponential combinations, it is difficult to understand the impact of each flag and to manually choose a good flag configuration for a given program. To reduce this human effort, numerous

auto-tuning techniques that focus on flag selection have been proposed [2]–[9].

Compiler auto-tuning is a typical search problem, i.e., searching a good flag configuration within the space of optimizations, and thus the existing techniques can be viewed as different ways of making the search problem easier. Traditional search-based techniques explore candidate configurations through hill climbing [3], genetic or evolutionary algorithms [5], [10], [11], random iterative optimization [12], automatic configuration tools such as irace [13], and multi-technique frameworks such as OpenTuner [14]. Since each candidate configuration must be compiled and executed, learning-based techniques have been proposed to reduce this cost by modeling the performance of the configuration and guiding the search. For example, BOCA uses Bayesian optimization to select promising compiler configurations under a limited tuning budget [4], COBAYN builds Bayesian networks to model relationships among program features, optimization flags, and performance [15], and CompTuner learns performance prediction models through multiple phases and uses particle swarm optimization to search for effective flag combinations [9]. More recent work improves search efficiency by exploiting structure in the optimization space, which can be classified as structure-aware techniques: CFSCA identifies program-relevant critical flags and searches more intensively over them [8], PDCAT uses compiler-documentation constraints and flag preferences to reduce and bias the search space [16], SRTuner mines synergistic relations among flags [17], and GroupTuner performs localized mutation over coherent option groups [18].

These techniques facilitate the searching process in different ways, but none of them address the critical problem in the searching process: *where should the search start for a target program, and can that starting point still be corrected during search?* In particular, the search-based techniques [13], [14] rely on generic or weakly informed starting points and may spend much of the online budget just reaching a useful region of the flag space. Learning-based techniques [4], [15] predict promising configurations, but even a strong predictor rarely gives the best one for every program in a single shot. Structure-aware techniques [8], [17] narrow the search space using mined or largely program-agnostic structure, yet still do not give each program its own starting distribution. The closest prior work to this question is PDCAT [16], which explicitly recognizes the importance of the initial search bias: it uses historical records to set initial flag-level probabilities and then uses this bias to guide subsequent search. However, this bias is still program-agnostic. Once the initial flag probabilities are determined, the same starting distribution is used for all target programs, regardless of their source-code characteristics. To sum up, existing priors are either generic, shared across programs, or fixed once online search begins. Runtime feedback may steer the search process under these priors, but it does not update a program-conditioned prior that generates a different starting distribution for each target program. As a result, programs with very different optimization preferences may still begin from similar regions of the flag space, wasting budget that a program-specific, adaptable policy could save.

Our key insight is twofold. First, historical tuning data, accumulated across many programs, reveal how program characteristics relate to promising regions of the flag space, and can therefore guide where the search should start for each particular program. Second, an offline starting point alone is not enough: a new target program may differ from those seen offline. Therefore, the tuner must keep adapting to the behavior observed on the target program, which naturally calls for online reinforcement learning driven by measured speedups over `-O3`.

Based on these observations, we propose an Offline-to-online Reinforcement LeArning framework for compiler auto-tuning (abbreviated as WarmTuner). The offline stage is designed to provide a customized starting point for each target program. Instead of using a shared initial configuration, a manually designed preference bias, or a one-shot prediction of the final best configuration, WarmTuner learns a program-conditioned flag-selection policy from historical tuning records. Given the source-code representation of a target program and the `-O3` reference state, the policy outputs flag-enable probabilities over the full flag space. These probabilities define a program-specific starting point, so different programs can begin online tuning from different regions that match their predicted optimization preferences. The online stage further refines this customized starting point with feedback from the same target program. WarmTuner samples candidate configurations from the program-specific policy, measures their speedups over `-O3`, and updates the same policy using the measured rewards. Therefore, the initial distribution is not only customized offline, but also correctable online when it mismatches the target program. This differs from methods whose starting bias may guide subsequent search but remains external to, or fixed relative to, the policy that generates future configurations.

We instantiate the online update with a reinforcement learning algorithm, Group Relative Policy Optimization (GRPO) [19]. In each online round, WarmTuner samples a group of candidate flag configurations from the current policy, compiles and executes them on the target program, and uses their measured speedups over `-O3` as rewards. The policy is then updated according to the relative performance of candidates in the same group, so that later rounds sample configurations from a policy already adapted to the target program. This process repeats until the tuning budget is exhausted, and WarmTuner returns the best measured flag configuration.

To evaluate WarmTuner, we conducted experiments on GCC 15.2.0 using two widely used compiler auto-tuning benchmarks, cBench [20] and PolyBench [21]. We compare WarmTuner with four representative compiler auto-tuning techniques, including RIO [12], SRTuner [17], GroupTuner [18], and PDCAT [16]. Under the 5,000s budget, the flag configurations generated by WarmTuner outperform the default `-O3` setting on all 30 selected evaluation programs, with speedups

ranging from $1.133\times$ to $4.159\times$ (i.e., 13.3%–315.9% improvement over `-O3`). Among the four compared techniques, WarmTuner achieves the highest average speedup ($1.732\times$) and significantly outperforms RIO, SRTuner, and PDCAT. Against the strongest baseline, GroupTuner, WarmTuner still obtains a higher average speedup ($1.732\times$ vs. $1.669\times$), more best results (14 vs. 11), and fewer worst results (1 vs. 4). To study the impact of its components, we construct 7 variants of WarmTuner; the ablation results show that the main components improve tuning performance.

The contributions of this paper are summarized as follows:

1) **An offline-to-online reinforcement learning framework for compiler auto-tuning**, which learns an initial flag selection policy from historical tuning data and improves it on the target program through online policy optimization with real execution feedback.
2) **An extensive experiment** on the latest version of GCC (i.e., GCC 15.2.0) using two widely used benchmarks, cBench and PolyBench, demonstrates the performance of WarmTuner as well as its components.
3) **A reproducible package**, available in a public repository [22].

II. Technique

Given a target program, compiler flag selection decides which optimization flags to enable or disable so that the compiled program runs faster than the default `-O3` setting. Many tuners provide useful search guidance, but this guidance is usually fixed once online measurement begins. WarmTuner instead represents the guidance as a trainable flag policy: offline pretraining initializes it from historical tuning data, and online tuning adapts it with feedback from the target program.

Figure 1 presents an overview of WarmTuner, which comprises two stages: (1) offline pretraining and (2) online tuning. The offline stage consumes historical tuning records to train a single program-conditioned flag policy that can be reused across target programs. For each new target program, this policy recommends a program-specific starting point in the flag space for online tuning. The online stage starts from this point and updates the same policy on the target program through real compile-run feedback.

A. Offline Pretraining

To give each target program its own starting point rather than a generic, program-agnostic one, WarmTuner uses the offline stage to turn historical tuning experience into a program-conditioned distribution over the flag space. To this end, WarmTuner casts offline learning as supervised policy pretraining over the full flag space, rather than treating historical data as a one-shot performance predictor or a fixed search bias as in prior work. The offline input consists of historical tuning records, each containing a source program, a measured binary flag configuration, and the corresponding speedup over `-O3`. These records are not tied to any single tuner: they can be collected from existing techniques such as RIO [12], PDCAT [16], or other search procedures. From these records, WarmTuner first represents each program from its source code and then uses good historical configurations as supervised signals to pretrain a flag policy with per-flag binary cross-entropy.

a) Program Representation.: To produce a program-specific starting point, WarmTuner first converts each source program into an embedding. This embedding is later passed to the flag policy so that flag probabilities depend on the target program.

WarmTuner builds this embedding from source code using CodeBERT. This design avoids relying on GCC-specific IR features or pass traces, and keeps feature extraction independent of compiler internals. Source code exposes structures that are relevant to optimization decisions, such as loops, branches, function calls, and memory accesses. A pretrained code model provides a compact way to encode these source-level signals.

Since a complete program may exceed the input length supported by CodeBERT, WarmTuner first concatenates the C source files of the program and splits the text into bounded-length chunks. Each chunk is tokenized and truncated to at most 512 tokens [23]. WarmTuner encodes each chunk with CodeBERT and uses the hidden state of the `[CLS]` token as the chunk embedding.

To obtain a single embedding for the whole program, WarmTuner averages the chunk embeddings:

$$c_p = \frac{1}{m}\sum_{j=1}^{m} h_j. \tag{1}$$

Here, $h_j \in \mathbb{R}^{768}$ denotes the embedding of the j-th chunk, and m is the number of chunks for program p. Averaging chunk embeddings gives WarmTuner a program embedding c_p regardless of program size. This embedding is used as the program input to the flag policy described next.

b) Supervised Pretraining.: Given the program embedding c_p, WarmTuner trains a program-conditioned flag policy to estimate the enable probability of each optimization flag for program p. The policy also takes the `-O3` flag state x_0 as input, since WarmTuner searches for configurations that improve over the compiler's default `-O3` setting. The policy maps the program representation and the `-O3` reference state to per-flag logits.

In our implementation, the same policy architecture is used in both offline pretraining and online tuning. Let n be the number of optimization flags considered by WarmTuner. The `-O3` flag vector is first encoded by a learnable embedding layer. For each optimization flag, the model maintains two 16-dimensional embeddings, corresponding to the disabled and enabled states, respectively. For each flag, the embedding corresponding to its `-O3` state is selected, and the resulting flag-state embeddings are stacked and flattened. A linear layer with ReLU activation then maps the flattened vector to a 128-dimensional state representation. This state representation is concatenated with the program embedding and passed through a multilayer perceptron with three linear layers of dimensions $896 \to 512 \to 256 \to 16n$, where LayerNorm and ReLU

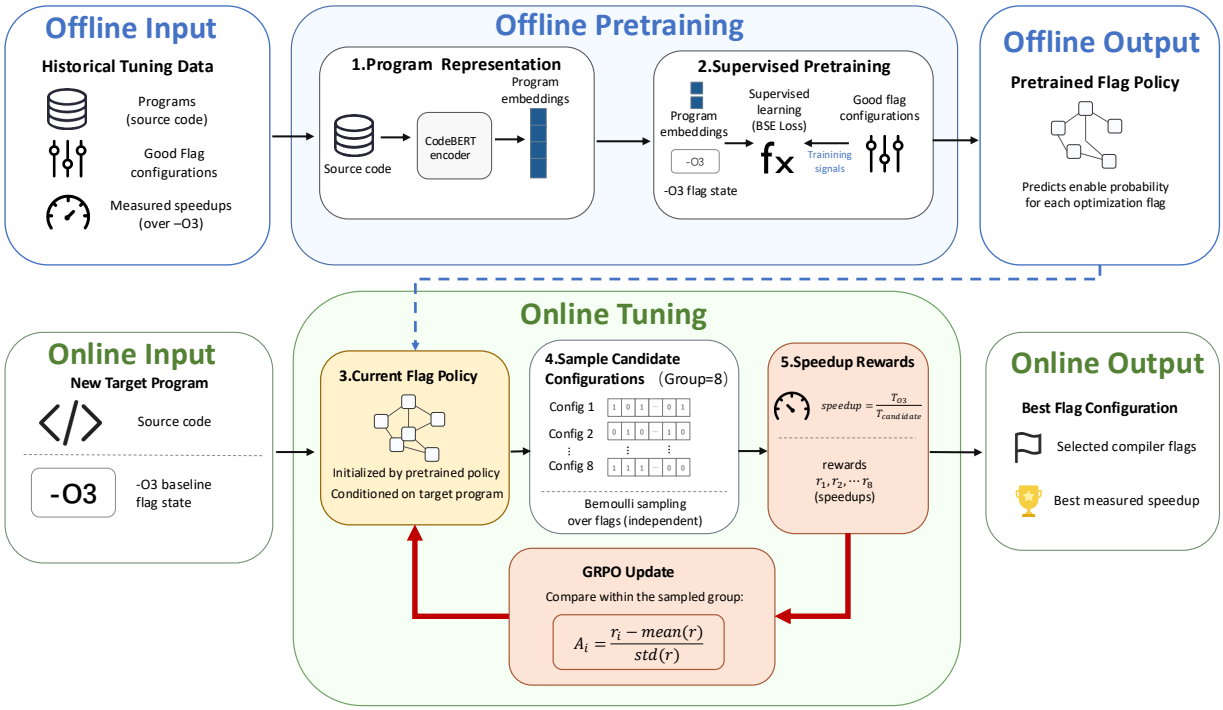

Fig. 1. Overview of WarmTuner.

are applied after the first two linear layers. The output is reshaped into an $n \times 16$ matrix, so that each optimization flag is associated with one 16-dimensional prediction vector. Finally, the logit z_i of each flag is computed as the dot product between its 16-dimensional prediction vector and a trainable `-O3` anchor embedding. The enable probability of flag $i \in \{1, \ldots, n\}$ is then computed as

$$q_i = \pi_\theta(x_i = 1 \mid c_p, x_0) = \sigma(z_i), \tag{2}$$

where θ denotes the policy parameters, $x_i \in \{0, 1\}$ is the binary decision for the i-th flag, and $\sigma(\cdot)$ is the sigmoid function. Here, $x_i = 1$ means enabling the flag and $x_i = 0$ means disabling it.

To construct supervision, WarmTuner selects historical configurations that improve over `-O3` and treats their binary flag vectors as labels. Given a training pair (p, y), where $y = (y_1, \ldots, y_n)$ is such a label vector for program p, the policy is trained with per-flag binary cross-entropy:

$$\mathcal{L}_{\text{pre}}(\theta) = -\sum_{i=1}^{n} \left[y_i \log q_i + (1 - y_i) \log(1 - q_i) \right]. \tag{3}$$

Here, y_i is the historical label for the i-th flag.

The resulting pretrained policy provides the program-specific starting point for online tuning.

B. Online Tuning

Because the effects of optimization flags are program-dependent, the policy learned from historical data may not fully match a new target program. WarmTuner therefore refines the policy online using real compile-run feedback. We instantiate this online update with GRPO because each tuning round evaluates a group of candidate configurations for the same program, making their relative performance directly comparable.

Since compiler flag tuning decides whether each optimization flag should be enabled or disabled, WarmTuner models a complete configuration as a binary decision vector. Let $x = (x_1, \ldots, x_n)$ denote a candidate flag configuration, where n is the number of optimization flags and $x_i \in \{0, 1\}$ indicates whether the i-th flag is enabled. Given the target program embedding c_p and the `-O3` flag state x_0, the policy parameterized by θ outputs an enable probability q_i for each flag. These probabilities define a factorized Bernoulli policy:

$$\pi_\theta(x \mid c_p, x_0) = \prod_{i=1}^{n} q_i^{x_i} (1 - q_i)^{1 - x_i}, \tag{4}$$

where $\pi_\theta(x \mid c_p, x_0)$ is the probability of sampling the complete configuration x under the current policy.

In each online round, WarmTuner samples K configurations $\{x^{(1)}, \ldots, x^{(K)}\}$ from the current policy. Each sampled binary vector is translated into compiler options: bit 1 enables the corresponding `-f...` flag, while bit 0 uses the `-fno-...` form.

WarmTuner then compiles and executes the target program with each candidate configuration. The reward is defined as speedup over `-O3`, with failed measurements penalized. For candidate $x^{(k)}$, let $T^{(k)}$ be its runtime, T_{O3} be the runtime

under `-O3`, and $m^{(k)}$ indicate whether compilation and execution succeed. The reward is

$$r^{(k)} = \begin{cases} T_{O3}/T^{(k)}, & m^{(k)} = 1, \\ -1, & m^{(k)} = 0. \end{cases} \tag{5}$$

Thus, $r^{(k)} > 1$ means the candidate is faster than `-O3`, while failed candidates receive reward -1.

Since all K candidates in a round are evaluated on the same program with the same metric, their rewards can be compared directly. WarmTuner uses this group-relative comparison to compute the advantage without training a separate value model:

$$A^{(k)} = \frac{r^{(k)} - \mathrm{mean}(r^{(1)}, \ldots, r^{(K)})}{\mathrm{std}(r^{(1)}, \ldots, r^{(K)}) + \delta}, \tag{6}$$

where $A^{(k)}$ is the advantage of candidate $x^{(k)}$, and δ is a small constant for numerical stability. A positive value means the candidate performs better than the group average, while a negative value means the opposite.

The policy should move toward candidates with positive advantages, but each update is based on only one measured group and should therefore be constrained. We use a clipped GRPO/PPO-style objective [19], [24]. Let θ_{old} be the policy parameters used to sample the current group. For candidate $x^{(k)}$, the probability ratio between the current policy and the sampling policy is

$$\rho^{(k)}(\theta) = \frac{\pi_\theta(x^{(k)} \mid c_p, x_0)}{\pi_{\theta_{\mathrm{old}}}(x^{(k)} \mid c_p, x_0)}. \tag{7}$$

To keep the objective readable, we first define the clipped policy-improvement term for each candidate:

$$\begin{aligned} g^{(k)}(\theta) = \min \big(& \rho^{(k)}(\theta) A^{(k)}, \\ & \mathrm{clip}(\rho^{(k)}(\theta), 1-\epsilon, 1+\epsilon) A^{(k)} \big). \end{aligned} \tag{8}$$

Here, ϵ is the clipping threshold.

The online loss aggregates this clipped signal over the sampled group and adds entropy regularization:

$$\mathcal{L}_{\mathrm{online}}(\theta) = -\frac{1}{K} \sum_{k=1}^{K} g^{(k)}(\theta) - \beta H(\pi_\theta). \tag{9}$$

Here, $H(\pi_\theta)$ is the entropy of the Bernoulli policy, and β controls the entropy bonus. Minimizing this loss increases the probability of better-than-average configurations while clipping limits overly large policy changes.

Online tuning repeats this process until the tuning budget is exhausted. WarmTuner returns the best measured configuration x_{best} and its speedup r_{best}.

III. Experimental Setup

To evaluate the performance of WarmTuner, this experiment is designed to answer two research questions.

- **RQ1: Effectiveness.** How does WarmTuner perform compared to representative compiler auto-tuning techniques? This RQ investigates the performance of flag configurations selected by different compiler auto-tuning techniques.
- **RQ2: Ablation analysis.** How do the main components and design choices of WarmTuner contribute to the performance? This RQ investigates the impact of offline pretraining, program representation, the choice of online policy optimizer, GRPO group size, and different historical tuning data sources.

TABLE I
Statistics of Programs from cBench and PolyBench.

ID	Program	Lines	ID	Program	Lines
C1	automotive_bitcount	954	C11	consumer_jpeg_d	26,183
C2	automotive_susan_e	2,129	C12	consumer_tiff2bw	22,271
C3	automotive_susan_c	2,129	C13	consumer_tiffdither	22,184
C4	automotive_susan_s	2,129	C14	consumer_tiffmedian	22,756
C5	consumer_tiff2rgba	22,321	C15	network_dijkstra	199
C6	consumer_jpeg_c	26,950	C16	network_patricia	634
C7	security_sha	297	C17	security_blowfish_d	1,524
C8	bzip2e	7,200	C18	security_blowfish_e	1,524
C9	telecom_adpcm_c	389	C19	telecom_CRC32	307
C10	bzip2d	7,200	C20	automotive_qsort1	227
P1	correlation	248	P16	syrk	199
P2	covariance	218	P17	trmm	199
P3	symm	231	P18	atax	198
P4	2mm	252	P19	bicg	214
P5	3mm	267	P20	doitgen	203
P6	cholesky	212	P21	mvt	210
P7	lu	210	P22	durbin	195
P8	nussinov	569	P23	gramschmidt	220
P9	heat-3d	211	P24	ludcmp	247
P10	jacobi-2d	200	P25	trisolv	183
P11	gemm	221	P26	deriche	265
P12	gemver	249	P27	floyd-warshall	176
P13	gesummv	210	P28	adi	237
P14	seidel-2d	179	P29	fdtd-2d	245
P15	syr2k	214	P30	jacobi-1d	186

A. Dataset

Following existing works [8], [16]–[18], we use GCC 15.2.0, the latest GCC release available at the time the experiments were conducted, as the target compiler, and evaluate WarmTuner on two widely used C benchmarks, cBench [20] and PolyBench [21], which cover a wide range of practical functions ranging from 100 to 26,000 lines of code. For compiler flag tuning, we consider 126 optimization flags supported by GCC. Table I summarizes the target programs used in our experiments, where the "ID" column uses C1–C20 to denote the 20 cBench programs and P1–P30 to denote the 30 PolyBench programs.

B. Compared Techniques

To validate the effectiveness of our technique, we compare WarmTuner against four representative compiler auto-tuning techniques: (1) **RIO** [12], which is a random iterative optimization technique and is commonly used as a baseline in compiler auto-tuning studies, (2) **SRTuner** [17], which is a search-based auto-tuning technique that exploits synergistic relations among optimization flags to guide the search process, (3) **GroupTuner** [18], which is a group-aware technique that

organizes flags into coherent option groups and performs localized mutation around historically good configurations, and (4) **PDCAT** [16], which is a preference-driven compiler auto-tuning technique that reduces the search space using constraints and preferences. In particular, GroupTuner and PDCAT are among the most recently published techniques and represent the current state of the art in flag selection.

For a fair comparison, we use a time budget as the termination condition of the compared techniques and compare the flag configurations generated by these techniques. We report results at 1,250, 2,500, and 5,000 seconds, respectively, to compare their performance. Moreover, we repeat each technique five times to alleviate the influence of randomness resulting from the compiler auto-tuning techniques and use the median result to denote the performance of an auto-tuning technique.

C. Implementation

We implement WarmTuner in Python based on PyTorch and Hugging Face Transformers [25], [26].

For offline pretraining, the learning rate is set to 5×10^{-4}, the batch size is set to 64, and the model is trained for 300 epochs. We use AdamW with weight decay 10^{-4} and CosineAnnealingLR with $\eta_{\min} = 10^{-5}$. Gradient clipping is applied with $\texttt{max\_norm} = 1.0$ during offline pretraining. In particular, the offline pretraining stage uses historical tuning data collected by random iterative optimization. After filtering configurations whose speedup is not larger than 1, the three training splits contain 29,004, 25,525, and 21,553 records, respectively.

In the GRPO instantiation of online tuning, the group size K is set to 8, the online learning rate is set to 10^{-4}, the entropy coefficient β is set to 0.02, and the clipping threshold ϵ is set to 0.2. The numerical stability constant δ in advantage normalization is set to 10^{-8}. Each online round performs one clipped policy update and applies gradient clipping with $\texttt{max\_norm} = 0.5$.

Following the protocol in previous work [16], we perform three independent random splits of the programs. In each split, 70% of the total programs (i.e., 35 programs) are used for offline pretraining and the remaining 30% serve as testing programs for evaluation. We train one model for each split and evaluate it on the corresponding testing programs to reduce the influence of a particular train-test selection.

For the compared techniques, we directly use their official reproducible package when available (i.e., for SRTuner, GroupTuner and PDCAT); otherwise, we re-implement the technique (i.e., RIO) strictly following the description in the paper. The experiments are performed on a server with two Intel Xeon Platinum 8481C CPUs, 1.0 TiB memory, one NVIDIA GeForce RTX 4090 D GPU, and Ubuntu 22.04.3 LTS.

D. Measurement

Following prior work [8], [16]–[18], we report speedup over GCC `-O3`. GCC also provides `-O1`, `-O2`, and `-Ofast`, but `-O3` is the highest standard optimization level and is commonly used as the baseline in compiler tuning studies.

To obtain the runtime performance improvement provided by a compiler auto-tuning technique, we compile any given program P using both the selected optimization sequence from the auto-tuning technique and the default `-O3` optimization. This process yields two compiled programs P_s and P_d. Then we execute both programs and measure the speedup of P by dividing the execution time of P_d (i.e., compiled with `-O3`) by the execution time of P_s (i.e., compiled with the selected optimization sequence). A speedup larger than 1 indicates that the selected optimization sequence outperforms `-O3`. We use this result to measure compiler auto-tuning techniques.

To reduce measurement noise, WarmTuner measures each candidate configuration and the `-O3` baseline in the same worker and in adjacent time windows. Each worker uses an independent temporary directory that isolates intermediate files during the parallel evaluation of candidates.

IV. Experimental Results

A. Overall Effectiveness (RQ1)

To answer RQ1, we compare WarmTuner with representative compiler auto-tuning techniques under the same tuning budgets, using speedup over GCC `-O3` as the metric. Table II presents the program-wise speedups achieved by the compared techniques. Each cell reports the speedup over `-O3`, where a larger value indicates better runtime performance. For each target program, the best result is highlighted in bold and the worst is underlined. When a program appears in multiple train-test selections, the table reports its average speedup and the number of appearances.

As shown in the table, WarmTuner improves over GCC `-O3` on all target programs, with the speedup ranging from $1.133\times$ to $4.159\times$ in 5,000 seconds. Its performance also improves as the tuning budget increases, suggesting that online tuning continues to refine the configurations sampled from the pretrained policy.

1) Statistics Analysis: To further examine whether WarmTuner outperforms the compared baselines, we statistically analyze the results from five perspectives. First, we count how often each baseline performs worse than WarmTuner under different tuning budgets. Second, we count the number of best and worst results achieved by each technique, which reflects both effectiveness and robustness. Third, we compare the average speedup across all testing programs to measure the overall acceleration performance. Fourth, we conduct paired statistical tests between WarmTuner and each baseline. Finally, we report the offline pretraining cost introduced by WarmTuner.

Comparing WarmTuner against each baseline program by program, we find that most baselines produce less effective flag configurations under the same tuning budget. Specifically, RIO, SRTuner, GroupTuner, and PDCAT perform worse than WarmTuner on 27/27/26, 24/25/25, 20/21/18, and 21/18/19 of the 30 programs in 1,250/2,500/5,000 seconds, respectively.

We further count, for each case, how often each technique achieves the best result (in **bold**) and the worst result

TABLE II
SPEEDUP OVER -O3 ACHIEVED BY COMPARED TECHNIQUES.

Technique	ID	1,250s	2,500s	5,000s	ID	1,250s	2,500s	5,000s	ID	1,250s	2,500s	5,000s	ID	1,250s	2,500s	5,000s	ID	1,250s	2,500s	5,000s
WarmTuner		1.700	1.734	1.759		1.525	1.541	1.585		1.400	1.425	1.458		1.251	1.336	1.392		1.334	1.438	1.518
RIO		1.512	1.571	1.606		1.417	1.465	1.545		1.347	1.399	1.485		1.420	1.420	1.420		1.108	1.207	1.323
SRTuner	C1 (3)	1.796	1.839	1.863	C2 (1)	1.581	1.581	1.662	C3 (1)	1.414	1.414	1.414	C5 (1)	1.178	1.222	1.222	C6 (3)	1.074	1.137	1.214
GroupTuner		**1.856**	**1.896**	**1.962**		1.525	1.721	1.723		1.300	1.377	1.402		**1.442**	**1.541**	**1.618**		1.401	**1.576**	**1.624**
PDCAT		1.853	1.884	1.896		**1.671**	**1.799**	**1.799**		**1.559**	**1.564**	**1.564**		1.313	1.503	1.575		**1.470**	1.515	1.570
WarmTuner		2.059	2.153	2.153		**1.500**	**1.516**	**1.564**		**3.012**	**3.090**	3.162		**1.458**	**1.458**	1.494		1.444	1.460	**1.623**
RIO		2.116	2.279	2.376		1.270	1.382	1.442		2.661	2.661	2.883		1.254	1.323	1.323		1.386	1.386	1.461
SRTuner	C7 (1)	2.017	2.035	2.320	C8 (3)	1.241	1.284	1.299	C9 (1)	2.565	2.623	2.623	C12 (1)	1.167	1.167	1.167	C14 (1)	1.297	1.307	1.307
GroupTuner		2.178	**2.489**	**2.638**		1.425	1.435	1.487		2.538	2.894	**3.375**		1.400	1.427	**1.618**		1.371	1.371	1.464
PDCAT		**2.319**	2.319	2.481		1.384	1.400	1.518		1.578	1.602	1.699		1.324	1.427	1.599		**1.486**	**1.510**	1.510
WarmTuner		1.553	1.586	**1.667**		**1.883**	**1.974**	**2.047**		1.692	**1.723**	**1.888**		**1.224**	1.251	1.284		**1.224**	1.235	1.249
RIO		1.315	1.440	1.528		1.651	1.672	1.732		1.600	1.666	1.671		1.196	1.221	1.243		1.209	1.210	1.226
SRTuner	C15 (1)	1.439	1.439	1.489	C18 (2)	1.494	1.494	1.500	C20 (1)	1.530	1.530	1.587	P1 (1)	1.174	1.203	1.203	P2 (2)	1.139	1.220	1.220
GroupTuner		1.483	1.498	1.618		1.492	1.606	1.760		**1.706**	1.713	1.719		1.044	1.064	1.165		1.055	1.058	1.110
PDCAT		**1.598**	**1.614**	1.621		1.655	1.684	1.859		1.690	1.706	1.706		1.190	**1.317**	**1.317**		1.206	**1.248**	**1.290**
WarmTuner		**1.525**	**1.532**	**1.570**		**1.272**	**1.272**	**1.289**		**1.866**	**1.941**	**2.233**		1.391	1.391	1.509		**1.398**	**1.402**	**1.402**
RIO		1.128	1.128	1.165		1.121	1.147	1.147		1.202	1.202	1.229		1.336	1.336	1.373		1.221	1.227	1.278
SRTuner	P4 (1)	1.090	1.133	1.133	P5 (1)	1.141	1.141	1.175	P11 (1)	1.195	1.204	1.207	P12 (1)	**1.462**	**1.462**	1.470	P15 (1)	1.225	1.249	1.249
GroupTuner		1.151	1.189	1.241		1.107	1.142	1.142		1.285	1.298	1.352		1.418	1.461	**1.531**		1.137	1.137	1.255
PDCAT		1.116	1.197	1.244		1.108	1.108	1.124		1.365	1.365	1.365		1.342	1.342	1.394		1.169	1.184	1.194
WarmTuner		**1.483**	**1.507**	**1.512**		**1.284**	**1.296**	**1.313**		1.578	1.657	1.675		**1.668**	**1.700**	**1.828**		**1.737**	**1.780**	1.835
RIO		1.237	1.237	1.304		1.220	1.254	1.290		1.558	1.567	1.583		1.384	1.384	1.454		1.592	1.652	1.755
SRTuner	P16 (1)	1.190	1.202	1.202	P17 (2)	1.228	1.248	1.248	P20 (2)	1.670	1.680	1.682	P21 (1)	1.591	1.649	1.649	P23 (3)	1.621	1.670	1.676
GroupTuner		1.206	1.228	1.376		1.169	1.227	1.239		1.651	1.706	**1.767**		1.567	1.595	1.811		1.610	1.697	**1.860**
PDCAT		1.192	1.210	1.370		1.208	1.242	1.275		**1.716**	**1.717**	1.747		1.478	1.478	1.478		1.664	1.693	1.754
WarmTuner		1.368	1.383	**1.416**		1.812	2.054	2.054		1.098	1.116	1.133		1.148	1.163	1.192		**3.583**	**3.712**	**4.159**
RIO		1.273	1.299	1.299		1.473	1.566	1.584		**1.099**	1.133	1.155		1.068	1.068	1.131		2.867	2.932	3.107
SRTuner	P24 (1)	1.200	1.200	1.200	P26 (1)	**2.224**	**2.224**	**2.233**	P27 (3)	1.070	1.087	1.111	P29 (1)	1.102	1.126	1.126	P30 (2)	1.777	1.895	1.971
GroupTuner		**1.390**	**1.390**	1.392		1.398	1.434	1.644		1.094	1.113	**1.161**		**1.154**	**1.238**	**1.258**		3.394	3.628	3.763
PDCAT		1.178	1.238	1.288		1.312	1.317	1.436		1.095	**1.140**	1.142		1.083	1.111	1.166		3.022	3.234	3.410

TABLE III
NUMBER OF BEST AND WORST PERFORMANCE RESULTS.

Best results/Worst results			
Technique	**1,250s**	**2,500s**	**5,000s**
WarmTuner	**15/0**	**14/0**	**14/1**
RIO	1/8	0/9	0/5
SRTuner	2/12	2/13	1/16
GroupTuner	5/7	6/5	11/4
PDCAT	7/3	8/3	4/4

(underlined). As summarized in Table III, WarmTuner obtains the largest number of best results at all three budgets (15/14/14 under 1,250s/2,500s/5,000s) and rarely the worst (only 0/0/1). In contrast, RIO and SRTuner produce more worse results (8/9/5 and 12/13/16, respectively). GroupTuner and PDCAT are more competitive, but still give more worst cases than WarmTuner (7/5/4 and 3/3/4). These counts show that WarmTuner performs consistently across programs, rather than excelling only in a small subset.

In terms of average speedup, WarmTuner also ranks first under all three budgets, reaching $1.616\times$, $1.661\times$, and $1.732\times$, respectively. Among the strongest baselines, GroupTuner reaches $1.498\times$, $1.572\times$, and $1.669\times$, while PDCAT reaches $1.478\times$, $1.522\times$, and $1.580\times$ under the same budgets. The other two baselines also remain lower; even at the largest budget, RIO and SRTuner reach only $1.537\times$ and $1.481\times$, respectively. These results show that WarmTuner consistently improves average tuning quality, with its advantage remaining visible even against the strongest competing tuners.

We further conduct a paired t-test on the 30 testing programs in 5,000 seconds. WarmTuner significantly outperforms RIO, SRTuner, and PDCAT, with p-values of 3.98×10^{-4}, 3.87×10^{-3}, and 2.88×10^{-2}, respectively. The improvement over GroupTuner, the strongest baseline, is not statistically significant ($p = 1.74 \times 10^{-1}$); nevertheless, WarmTuner still achieves a higher average speedup and more best (and fewer worst) results.

Since WarmTuner introduces an offline pretraining stage that the other techniques do not have, we therefore report the offline pretraining cost of WarmTuner to discuss whether this stage incurs a large additional cost. Across the three random splits, supervised pretraining takes 9.40, 9.70, and 8.44 minutes, with an average of 9.18 minutes. Moreover, this cost is paid only once per split, after which the trained model is reused for all testing programs in that split, so it is negligible compared with the online tuning budget.

To sum up, WarmTuner improves over GCC -O3 on all testing programs and achieves the best overall performance among the compared techniques in terms of average speedup and best-result counts. This demonstrates the effectiveness of combining offline supervised pretraining with online reinforcement-

learning-based policy optimization for compiler flag autotuning.

2) Case Study: To examine whether the best configuration selected by WarmTuner is consistent with the target program's characteristics, we inspect its tuning result on `jacobi-1d`. As shown in Table II, WarmTuner reaches a summarized speedup of $4.159\times$ on this program in 5,000 seconds. `jacobi-1d` is a stencil kernel dominated by repeated one-dimensional array updates, so loop and floating-point optimizations are likely to affect its runtime.

In the inspected tuning log, the best measured configuration achieves a speedup of $4.940\times$. Among its enabled flags, we focus on `-ftree-loop-vectorize`, `-funroll-loops`, and `-ffast-math`, because their roles in GCC correspond to loop vectorization, loop unrolling, and aggressive floating-point optimization, respectively [27].

TABLE IV
INFLUENCE OF SELECTED FLAGS IN THE TUNING LOG OF JACOBI-1D.

Flag	Avg. On	Avg. Off
`-ftree-loop-vectorize`	1.030	0.727
`-funroll-loops`	1.027	0.852
`-ffast-math`	1.026	0.861

As shown in Table IV, candidates enabling these three flags consistently achieve higher average speedups than those disabling them. The clearest gap appears for `-ftree-loop-vectorize`, whose average speedup is $1.030\times$ when enabled but only $0.727\times$ when disabled. Similar trends appear for the other two flags. Although some candidates without these flags still outperform `-O3`, the enabled group performs better on average, suggesting that their appearance in the best configuration is not accidental.

These observations are consistent with the structure of `jacobi-1d`, whose inner loops repeatedly update contiguous floating-point arrays. This case study therefore suggests that WarmTuner can select optimization flags whose roles match the computation pattern of the target program.

B. Ablation Analysis (RQ2)

To answer RQ2, we construct 7 variants of WarmTuner by removing or replacing each component individually or modifying parameters. We denote each variant as WarmTuner with a subscript indicating the modified component. We analyze the contribution of each component of WarmTuner by comparing the acceleration performance between WarmTuner and these variants in 1,250, 2,500, and 5,000 seconds.[1]

1) Offline Pretraining: To investigate the contribution of offline pretraining, we replace the supervised pretraining stage by starting online tuning from a randomly initialized policy, resulting in a variant denoted as WarmTuner$_{NoPre}$. Table V shows that WarmTuner performs better than WarmTuner$_{NoPre}$ on 15/15/13 out of 20 programs in 1,250/2,500/5,000 seconds. The average speedup gains are +0.069, +0.046, and +0.033. This result indicates that offline pretraining provides a useful initial policy, enabling online tuning to sample from more promising flag distributions before enough target-program feedback has been collected.

[1]In RQ1, we conduct three separate random selections of the initial tuning set to control the influence of randomness but achieve a consistent conclusion across different selections. In this ablation analysis, we use one selection result as initial tuning and testing programs to control the expensive evaluation costs resulting from the 7 variants. The other settings remain the same as in Section IV-A.

TABLE V
IMPACT OF OFFLINE PRETRAINING.

Technique	ID	1,250s	2,500s	5,000s	ID	1,250s	2,500s	5,000s
WarmTuner	C2	**1.525**	**1.541**	**1.585**	C3	**1.400**	**1.425**	**1.458**
WarmTuner$_{NoPre}$		1.440	1.500	1.578		1.335	1.385	1.452
WarmTuner	C5	1.251	1.336	**1.392**	C7	**2.059**	**2.153**	**2.153**
WarmTuner$_{NoPre}$		**1.313**	**1.359**	1.383		1.900	2.103	2.151
WarmTuner	C9	**3.012**	**3.090**	**3.162**	C12	**1.458**	1.458	1.494
WarmTuner$_{NoPre}$		2.977	3.037	3.151		1.366	**1.480**	**1.571**
WarmTuner	C14	**1.444**	**1.460**	1.623	C15	**1.553**	**1.586**	**1.667**
WarmTuner$_{NoPre}$		1.370	1.420	**1.819**		1.450	1.508	1.526
WarmTuner	C20	1.692	**1.723**	**1.888**	P1	1.224	**1.251**	**1.284**
WarmTuner$_{NoPre}$		**1.702**	1.713	1.713		**1.240**	1.246	1.250
WarmTuner	P4	**1.525**	**1.532**	1.570	P5	1.272	1.272	1.289
WarmTuner$_{NoPre}$		1.463	1.489	**1.583**		**1.282**	**1.282**	**1.304**
WarmTuner	P11	**1.866**	**1.941**	**2.233**	P12	**1.391**	1.391	**1.509**
WarmTuner$_{NoPre}$		1.750	1.880	2.225		1.386	**1.478**	1.478
WarmTuner	P15	**1.398**	**1.402**	1.402	P16	**1.483**	**1.507**	1.512
WarmTuner$_{NoPre}$		1.379	1.398	**1.477**		1.438	1.500	**1.573**
WarmTuner	P21	**1.668**	**1.700**	**1.828**	P24	**1.368**	**1.383**	**1.416**
WarmTuner$_{NoPre}$		1.528	1.547	1.567		1.301	1.341	1.414
WarmTuner	P26	**1.812**	**2.054**	**2.054**	P29	1.148	1.163	1.192
WarmTuner$_{NoPre}$		1.373	1.403	1.426		**1.171**	**1.387**	**1.411**

TABLE VI
IMPACT OF PROGRAM REPRESENTATION.

Technique	ID	1,250s	2,500s	5,000s	ID	1,250s	2,500s	5,000s
WarmTuner	C2	1.525	1.541	1.585	C3	1.400	1.425	1.458
WarmTuner$_{NoRep}$		**1.633**	**1.633**	**1.753**		**1.474**	**1.518**	**1.553**
WarmTuner	C5	1.251	1.336	1.392	C7	2.059	2.153	2.153
WarmTuner$_{NoRep}$		**1.334**	**1.372**	**1.433**		**2.178**	**2.178**	**2.186**
WarmTuner	C9	**3.012**	**3.090**	**3.162**	C12	**1.458**	**1.458**	1.494
WarmTuner$_{NoRep}$		2.796	2.836	2.937		1.435	1.454	**1.500**
WarmTuner	C14	**1.444**	1.460	**1.623**	C15	**1.553**	**1.586**	**1.667**
WarmTuner$_{NoRep}$		1.396	**1.468**	1.482		1.529	1.529	1.582
WarmTuner	C20	**1.692**	**1.723**	**1.888**	P1	1.224	1.251	1.284
WarmTuner$_{NoRep}$		1.586	1.675	1.694		**1.269**	**1.269**	**1.290**
WarmTuner	P4	**1.525**	**1.532**	**1.570**	P5	**1.272**	1.272	1.289
WarmTuner$_{NoRep}$		1.431	1.518	1.518		1.241	**1.277**	**1.320**
WarmTuner	P11	**1.866**	**1.941**	**2.233**	P12	**1.391**	**1.391**	**1.509**
WarmTuner$_{NoRep}$		1.760	1.930	2.215		1.377	1.385	1.388
WarmTuner	P15	**1.398**	**1.402**	**1.402**	P16	1.483	1.507	1.512
WarmTuner$_{NoRep}$		1.280	1.372	1.380		**1.507**	**1.520**	**1.542**
WarmTuner	P21	**1.668**	**1.700**	**1.828**	P24	**1.368**	**1.383**	**1.416**
WarmTuner$_{NoRep}$		1.557	1.687	1.691		1.276	1.353	1.397
WarmTuner	P26	**1.812**	**2.054**	**2.054**	P29	**1.148**	**1.163**	1.192
WarmTuner$_{NoRep}$		1.693	1.693	1.693		1.145	1.160	**1.225**

2) Program Representation: To examine the effect of program representation, we mask the CodeBERT source-code representation by replacing it with an all-zero vector of the same dimension, resulting in a variant denoted as WarmTuner$_{NoRep}$. In this variant, online search starts from the same initial flag distribution before program-specific feedback is observed. As reported in Table VI, WarmTuner wins on 14/12/11 out of 20 programs across the three budgets and keeps a higher average speedup. At 5,000s, for example, WarmTuner reaches 1.686 on average, compared with 1.639 for WarmTuner$_{NoRep}$. This gap indicates that source-code features help the policy assign different starting distributions to different programs, rather than using the same search bias for all targets.

3) Choice of Policy Optimizer: To investigate the contribution of the GRPO update, we replace it with two alternative online reinforcement learning algorithms while keeping the pretrained policy unchanged. The first variant, denoted as WarmTuner$_{Reinf}$, uses REINFORCE [28] to directly update the policy with measured speedups as rewards. The second variant, denoted as WarmTuner$_{PPO}$, uses PPO [24] to perform clipped policy updates with advantage estimates from a learned value model. Table VII presents the acceleration performance of WarmTuner, WarmTuner$_{Reinf}$, and WarmTuner$_{PPO}$ on the selected test programs, with the best result in each case highlighted in bold.

TABLE VII
IMPACT OF ONLINE POLICY OPTIMIZER.

Technique	ID	1,250s	2,500s	5,000s	ID	1,250s	2,500s	5,000s
WarmTuner	C2	1.525	1.541	1.585	C3	1.400	1.425	1.458
WarmTuner$_{Reinf}$		**1.582**	**1.599**	**1.746**		**1.487**	**1.632**	**1.632**
WarmTuner$_{PPO}$		1.520	1.536	1.580		1.395	1.420	1.453
WarmTuner	C5	1.251	1.336	1.392	C7	2.059	2.153	2.153
WarmTuner$_{Reinf}$		1.256	**1.444**	**1.468**		**2.165**	**2.350**	**2.496**
WarmTuner$_{PPO}$		**1.311**	1.357	1.396		1.936	2.064	2.165
WarmTuner	C9	**3.012**	**3.090**	**3.162**	C12	**1.458**	**1.458**	1.494
WarmTuner$_{Reinf}$		2.557	2.557	2.689		1.238	1.400	1.400
WarmTuner$_{PPO}$		2.827	2.999	3.117		1.447	1.454	**1.502**
WarmTuner	C14	1.444	1.460	**1.623**	C15	1.553	1.586	**1.667**
WarmTuner$_{Reinf}$		1.300	1.308	1.383		1.403	1.428	1.499
WarmTuner$_{PPO}$		**1.489**	**1.508**	1.565		**1.571**	**1.612**	1.635
WarmTuner	C20	**1.692**	**1.723**	**1.888**	P1	1.224	**1.251**	**1.284**
WarmTuner$_{Reinf}$		1.677	1.683	1.683		1.207	1.207	1.230
WarmTuner$_{PPO}$		1.652	1.652	1.665		**1.244**	1.244	1.257
WarmTuner	P4	**1.525**	**1.532**	**1.570**	P5	**1.272**	1.272	1.289
WarmTuner$_{Reinf}$		1.127	1.127	1.184		1.115	1.151	1.173
WarmTuner$_{PPO}$		1.470	1.525	1.525		1.252	**1.291**	**1.330**
WarmTuner	P11	1.866	1.941	2.233	P12	1.391	1.391	1.509
WarmTuner$_{Reinf}$		1.174	1.241	1.354		1.357	1.404	1.425
WarmTuner$_{PPO}$		**2.015**	**2.015**	**2.305**		**1.524**	**1.524**	**1.569**
WarmTuner	P15	**1.398**	**1.402**	1.402	P16	1.483	1.507	1.512
WarmTuner$_{Reinf}$		1.227	1.276	1.413		1.215	1.278	1.306
WarmTuner$_{PPO}$		1.354	1.399	**1.448**		**1.486**	**1.546**	**1.566**
WarmTuner	P21	**1.668**	**1.700**	**1.828**	P24	**1.368**	1.383	1.416
WarmTuner$_{Reinf}$		1.435	1.435	1.472		1.237	1.237	1.309
WarmTuner$_{PPO}$		1.550	1.611	1.611		1.361	**1.461**	**1.461**
WarmTuner	P26	1.812	**2.054**	**2.054**	P29	**1.148**	**1.163**	**1.192**
WarmTuner$_{Reinf}$		1.301	1.345	1.362		1.060	1.096	1.101
WarmTuner$_{PPO}$		**1.842**	2.043	2.049		1.143	1.158	1.187

Among the three online RL algorithms, GRPO offers the best overall trade-off in our setting. WarmTuner outperforms WarmTuner$_{Reinf}$ on 16/15/15 of 20 programs in 1,250/2,500/5,000 seconds, with average speedups higher by +0.171, +0.159, and +0.169—indicating that using raw measured rewards directly, as REINFORCE does, yields a weaker and noisier update signal for compiler auto-tuning. Against WarmTuner$_{PPO}$, WarmTuner wins on 12/12/11 of 20 programs: PPO is competitive but provides no clear advantage here and additionally requires a separate value model that GRPO avoids by relying on within-group comparisons among candidates evaluated on the same program. Overall, the offline-to-online framework is compatible with different online RL algorithms, but GRPO remains the preferred choice, as it matches the group structure of compiler tuning.

4) Generality Across Data Sources: To test the generality of WarmTuner across historical tuning data sources, we pretrain the policy with records collected by different tuning techniques. The main method in this paper, WarmTuner, is pretrained with records collected by random iterative optimization (RIO), while we further construct a variant, denoted as WarmTuner$_{PDCAT}$, using records collected by PDCAT. WarmTuner$_{PDCAT}$ outperforms PDCAT on 25/21/22 out of 30 programs in 1,250/2,500/5,000 seconds, with average speedup gains of +0.123, +0.114, and +0.122. Meanwhile, WarmTuner outperforms RIO on 27/27/26 out of 30 programs, with gains of +0.174, +0.180, and +0.195. These results suggest that the framework can reuse historical records from different tuning strategies, rather than depending on one particular data collection method.

5) Effect of Group Size: To investigate the effect of GRPO group size, we construct two variants of WarmTuner by changing the number of candidates sampled and compared in each online tuning round while keeping the same testing programs, pretrained policy, and tuning budgets. The first variant, denoted as WarmTuner$_{K=4}$, uses $K = 4$ candidates per round, while the second variant, denoted as WarmTuner$_{K=16}$, uses $K = 16$. We compare these two variants with the default WarmTuner using $K = 8$ to study how the number of within-round comparisons affects online tuning performance.

Figure 2 shows the average speedup under the three tuning budgets, with the exact values annotated in the figure. Overall, $K = 4$ consistently performs the worst, suggesting that a small group provides too few within-round comparisons and leads to noisier advantage estimates. Increasing the group size to $K = 8$ clearly improves performance across all budgets. Further increasing it to $K = 16$ gives comparable performance at 1,250 seconds and only slightly higher speedups under larger budgets, indicating diminishing returns as the group grows. Since $K = 16$ doubles the per-round evaluation cost compared with $K = 8$, we use $K = 8$ as the default setting in the main experiments, which balances advantage-estimate quality and evaluation efficiency.

Overall, the ablations support the main design of WarmTuner. Offline pretraining and program representation give the online stage a program-specific starting point, while the

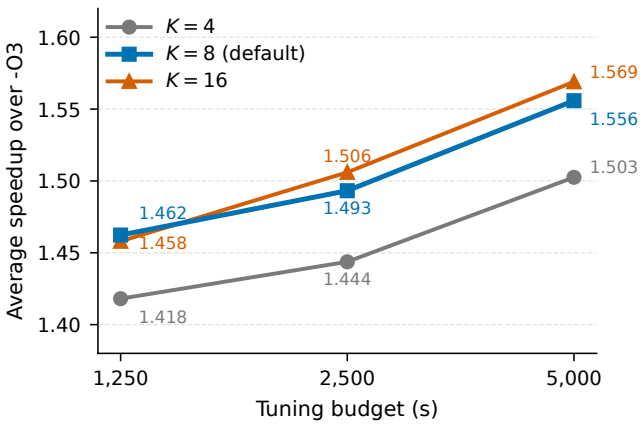

Fig. 2. Impact of GRPO group size on average speedup.

optimizer and group-size studies show how different online update choices affect performance under the same tuning budgets. The data-source study further shows that the same framework can work with records collected by different tuning strategies.

V. Threats to Validity

The internal threat mainly comes from the implementation of compiler auto-tuning techniques. To reduce this threat, we utilize the reproducible packages of the compared approaches and strictly re-implement these approaches based on their corresponding publications when reproducible packages are unavailable. In addition, the authors review the implementations to further reduce this threat.

The external threat mainly comes from the selected benchmarks, compiler, and hardware platform. To reduce this threat, we evaluate WarmTuner on cBench and PolyBench, which are commonly used in compiler auto-tuning works [8], [16]–[18]. We use GCC 15.2.0, the latest GCC release available at the time of our experiments, on an x86 server platform.

The construct threat mainly comes from the performance metrics. Following previous compiler auto-tuning works [8], [16]–[18], we use speedup over GCC `-O3` as the evaluation metric. Although GCC provides several optimization settings, including `-O1`, `-O2`, `-O3`, and `-Ofast`, `-Ofast` extends `-O3` by incorporating some unconventional optimizations that may violate certain international standards. In contrast, `-O1` and `-O2` do not activate many optimization flags. Consequently, `-Ofast` is generally not recommended in the literature, and studies on compiler tuning are typically compared against `-O3`.

VI. Discussion

Applicability beyond GCC flag selection. This work focuses on GCC optimization flag selection, but the proposed framework is not tied to GCC-specific mechanisms. It only requires an optimization interface with a well-defined action space and measurable feedback for each program. Under this abstraction, the same offline-to-online idea can be applied to other compilers and optimization tasks by redefining the actions and rewards. For example, LLVM pass ordering can use pass sequences as actions, inlining decisions can use optimization decisions as actions, and multi-objective tuning can define rewards based on execution time, code size, or their weighted combination. Exploring these extensions is left to future work.

Combination with existing tuning techniques. WarmTuner is designed as an offline-to-online framework rather than a method tied to a specific search algorithm. Existing compiler auto-tuning techniques can provide historical tuning records for offline pretraining, as long as the records contain evaluated configurations and their measured performance. The data-source study in Section IV shows that records collected by different tuning strategies can be converted into an online adaptable policy. This suggests that WarmTuner can reuse search experience accumulated by existing tuners and turn it into a program-specific starting point for further online refinement.

VII. Related Work

Existing work on compiler auto-tuning spans both optimization ordering and optimization selection [1]. To position this paper, we first briefly review representative phase-ordering studies and then focus on prior work most relevant to our setting, namely flag-selection techniques.

For phase ordering, MiCOMP uses optimization subsequences and machine learning to predict effective optimization orders [29]. AutoPhase applies deep reinforcement learning to compiler phase ordering for HLS [30]. Another machine-learning-based technique predicts method-specific optimization orders in JIT compilers [31]. These studies focus on optimization ordering, whereas this paper focuses on flag selection.

Compiler flag selection techniques mainly differ in how they explore or restrict the optimization space. Early work treats the compiler as a black-box evaluator and searches over measured configurations: random iterative optimization repeatedly samples configurations and keeps those that improve performance [12], genetic and evolutionary methods evolve a population of configurations through selection and mutation [5], [11], irace uses racing to discard weak configurations under a limited budget [13], and OpenTuner coordinates multiple search strategies in a common tuning framework [14]. Later work uses program features, historical measurements, or probabilistic models to guide configuration selection. Milepost GCC extracts program features and uses learned models to select optimization settings [32]; COBAYN builds Bayesian networks over program features and compiler optimizations to sample promising configurations [15]; BOCA uses Bayesian optimization to select configurations under limited measurements [4]; and CompTuner trains performance models in multiple phases and applies particle swarm optimization to search for effective flag combinations [9]. More recent techniques incorporate stronger structural guidance. CFSCA identifies

critical flags and searches the reduced space more intensively [8]; PDCAT derives constraints and preferences from compiler documentation to bias configuration generation [16]; SRTuner mines synergistic relations among flags to guide compatible flag combinations [17]; and GroupTuner organizes related options into groups and performs localized mutation around promising configurations [18]. Related learning-based work has also been studied in broader compiler optimization settings: CompilerGym provides reinforcement-learning environments for compiler optimization [33], MLGO trains learned policies for production LLVM decisions such as inlining and register allocation [34], and recent LLM-based approaches use large language models to predict, generate, or refine compiler optimization choices [35]–[38].

VIII. Conclusion

To alleviate human efforts on compiler tuning for a specific program, in the literature, several compiler auto-tuning techniques have been proposed, but all suffer from performance issues. In this paper, we proposed WarmTuner, an offline-to-online reinforcement learning framework for compiler flag auto-tuning. WarmTuner first learns a program-conditioned policy from historical tuning records, and then refines this policy on the target program using real compile-run feedback. In our implementation, GRPO serves as the default online optimizer by comparing groups of candidate configurations evaluated for the same program. According to the experimental results, WarmTuner achieves the best overall runtime performance among the compared compiler auto-tuning techniques. Moreover, the ablation analysis demonstrates the contribution of the main components in WarmTuner.

IX. Data Availability

We make the source code and data of WarmTuner available on website [22].

References

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

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

[3] 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.

[4] J. Chen, N. Xu, P. Chen, and H. Zhang, "Efficient compiler autotuning via bayesian optimization," in *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering*. IEEE, 2021, pp. 1198–1209.

[5] U. Garciarena and R. Santana, "Evolutionary optimization of compiler flag selection by learning and exploiting flags interactions," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, 2016, pp. 1159–1166.

[6] T. Kisuki, P. Knijnenburg, M. O'Boyle, and H. Wijshoff, *Iterative compilation in program optimization*. Universiteit Leiden, 2002.

[7] F. Li, F. Tang, and Y. Shen, "Feature mining for machine learning based compilation optimization," in *2014 Eighth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*. IEEE, 2014, pp. 207–214.

[8] M. Zhu and D. Hao, "Compiler auto-tuning via critical flag selection," in *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2023, pp. 1000–1011.

[9] M. Zhu, D. Hao, and J. Chen, "Compiler autotuning through multiple-phase learning," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 4, pp. 1–38, 2024.

[10] K. Hoste and L. Eeckhout, "COLE: Compiler optimization level exploration," in *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2008, pp. 165–174.

[11] S.-C. Lin, C.-K. Chang, and N.-W. Lin, "Automatic selection of GCC optimization options using a gene weighted genetic algorithm," in *Proceedings of the 13th Asia-Pacific Computer Systems Architecture Conference*. IEEE, 2008, pp. 1–8.

[12] 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*, vol. 9, no. 3, pp. 1–30, 2012.

[13] L. Pérez Cáceres, F. Pagnozzi, A. Franzin, and T. Stützle, "Automatic configuration of GCC using irace," in *International Conference on Artificial Evolution*. Springer, 2017, pp. 202–216.

[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, G. Mariani, G. Palermo, E. Park, J. Cavazos, and C. Silvano, "COBAYN: Compiler autotuning framework using bayesian networks," *ACM Transactions on Architecture and Code Optimization*, vol. 13, no. 2, pp. 1–25, 2016.

[16] M. Zhu, Z. Sun, and D. Hao, "PDCAT: Preference-driven compiler autotuning," *Proceedings of the ACM on Software Engineering*, vol. 2, no. FSE, pp. 847–867, 2025.

[17] S. Park, S. Latifi, Y. Park, A. Behroozi, B. Jeon, and S. Mahlke, "SRTuner: Effective compiler optimization customization by exposing synergistic relations," in *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE, 2022, pp. 118–130.

[18] B. Gao, M. Yao, Z. Wang, D. Liu, D. Li, X. Chen, and Y. Guo, "GroupTuner: Efficient group-aware compiler auto-tuning," in *Proceedings of the 26th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, 2025, pp. 122–133.

[19] Z. Shao, P. Wang, Q. Zhu, R. Xu, J. Song, X. Bi, H. Zhang, M. Zhang, Y. K. Li, Y. Wu *et al.*, "DeepSeekMath: Pushing the limits of mathematical reasoning in open language models," *arXiv preprint arXiv:2402.03300*, 2024.

[20] "cBench," https://ctuning.org/wiki/index.php/CTools:CBench, 2024, accessed: 2026-06-07.

[21] "PolyBench," https://sourceforge.net/p/polybench/wiki/Home/, 2024, accessed: 2026-06-07.

[22] Anonymous Authors, "Artifact Package for An Offline-to-Online Reinforcement Learning Framework for Compiler Autotuning," 2026. [Online]. Available: https://doi.org/10.5281/zenodo.21091417

[23] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, 2020, pp. 1536–1547.

[24] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[25] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "PyTorch: An imperative style, high-performance deep learning library," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[26] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz *et al.*, "Transformers: State-of-the-art natural language processing," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, 2020, pp. 38–45.

[27] GNU Compiler Collection (GCC), *Using the GNU Compiler Collection (GCC): Options That Control Optimization*, Free Software Foundation, 2025, [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc-15.2.0/gcc/Optimize-Options.html. Accessed: Jun. 23, 2026.

[28] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine learning*, vol. 8, no. 3, pp. 229–256, 1992.

[29] 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.

[30] Q. Huang, A. Haj-Ali, W. Moses, J. Xiang, I. Stoica, K. Asanovic, and J. Wawrzynek, "AutoPhase: Compiler phase-ordering for HLS with deep reinforcement learning," in *Proceedings of the 27th Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2019, pp. 308–308.

[31] S. Kulkarni and J. Cavazos, "Mitigating the compiler optimization phase-ordering problem using machine learning," in *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, 2012, pp. 147–162.

[32] 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, no. 3, pp. 296–327, 2011.

[33] 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 *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE, 2022, pp. 92–105.

[34] M. Trofin, Y. Qian, E. Brevdo, Z. Lin, K. Choromanski, and D. Li, "MLGO: A machine learning guided compiler optimizations framework," *arXiv preprint arXiv:2101.04808*, 2021.

[35] C. Cummins, V. Seeker, D. Grubisic, M. Elhoushi, Y. Liang, B. Roziere, J. Gehring, F. Gloeckle, K. Hazelwood, G. Synnaeve *et al.*, "Large language models for compiler optimization," *arXiv preprint arXiv:2309.07062*, 2023.

[36] C. Cummins, V. Seeker, D. Grubisic, B. Roziere, J. Gehring, G. Synnaeve, and H. Leather, "LLM Compiler: Foundation language models for compiler optimization," in *Proceedings of the 34th ACM SIGPLAN International Conference on Compiler Construction*, 2025, pp. 141–153.

[37] T. Cui, P.-C. Yew, S. McCamant, and A. Zhai, "DeCOS: Data-efficient reinforcement learning for compiler optimization selection ignited by LLM," in *Proceedings of the 39th ACM International Conference on Supercomputing*, 2025, pp. 943–958.

[38] X. Fang, J. Kang, R. Rocha, S. Ainsworth, and L. Mukhanov, "LLM-VeriOpt: Verification-guided reinforcement learning for LLM-based compiler optimization," in *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE, 2026, pp. 740–755.