Structure-Aware Compiler Auto-Tuning

Anderson Faustino da Silva
Universidade Estadual de Maringá
Maringá, Brazil
anderson.faustino_da_silva@tu-dresden.de

Christian Schwarz
TU Dresden
Dresden, Germany
christian.schwarz1@tu-dresden.de

Jeronimo Castrillon
TU Dresden
Dresden, Germany
jeronimo.castrillon@tu-dresden.de

Abstract

Modern compilers such as GCC expose large sets of optimization flags whose interactions induce a high-dimensional configuration space, making manual tuning impractical and exhaustive auto-tuning infeasible. Recent approaches, including Bayesian-optimization-based search and critical-flag selection via static analysis, improve sample efficiency but can exhibit premature convergence to locally optimal configurations. We present SACT, a structure-aware compiler auto-tuning framework that integrates static program characterization with surrogate-guided search to reduce stagnation. SACT extracts lightweight structural features, initializes a program embedding, and uses surrogate-sensitivity-based relevance estimates to adaptively focus the search on influential flags while maintaining controlled exploration. The embedding is refined online from compile–execute measurements, enabling the surrogate to capture program-specific flag interactions over the course of tuning. Experiments on GCC 15.2.0 using the PolyBench and MiBench benchmark suites show that SACT achieves a better geometric average speedup than prior methods while requiring a low wall-clock budget.

Keywords

Compiler autotuning, Optimization flags, Iterative compilation, Surrogate-guided search, Program structure, Dimensionality reduction

1 Introduction

Modern compilers such as GCC and LLVM expose large sets of optimization flags that control the transformations applied during compilation. These flags affect a wide range of behaviors, including vectorization, inlining, branch optimization, and memory-access transformations. A particular enabled/disabled assignment of these flags defines an *optimization-flag configuration*. With n binary flags, the induced search space contains 2^n configurations; for GCC 15.2.0, n exceeds 100, yielding more than 2^{100} possible configurations and rendering exhaustive evaluation infeasible [34].

Compilers provide default optimization levels such as `-O2` and `-O3`, which enable predefined flag subsets designed as conservative, general-purpose heuristics. Prior work has shown that program-specific optimization-flag configurations can outperform these defaults on realistic workloads [3, 31]. Efficiently identifying such configurations is a common goal in *compiler auto-tuning*.

Compiler flag auto-tuning has been addressed using both search-based and learning-based techniques. Classical search methods (e.g., genetic algorithms [16, 22] and hill climbing [10]) treat flag selection as a combinatorial optimization problem guided by compile–execute feedback, but can require many costly evaluations and scale poorly with increasing dimensionality. Surrogate-based methods instead reduce evaluation cost by predicting performance for unevaluated configurations. Bayesian optimization methods [5, 23] balance exploration and exploitation using an acquisition function, often requiring fewer evaluations than random or evolutionary baselines. BOCA [9] improves sample efficiency by combining a Random Forest surrogate (RFSM) with a decay-based acquisition strategy. In parallel, Critical-flag Selection via Static Analysis (CFSCA) [34] reduces dimensionality *a priori* by identifying critical flags using program-structure analysis and compiler documentation, then optimizing within the reduced subspace.

Despite these advances, premature convergence to local optima remains a recurring limitation. BOCA can converge quickly to a locally optimal region when trained solely on target-program measurements, with limited mechanisms to promote escape. CFSCA improves efficiency via flag filtering, but the resulting static boundary can exclude configurations that require flags outside the selected subset. In both cases, stagnation can prevent the search from reaching higher-quality configurations within a fixed budget.

This paper introduces SACT, a *structure-aware* auto-tuning method that uses program structural information to guide surrogate-based search while preserving controlled exploration. The central hypothesis is that structural characteristics (e.g., loop structure, branch behavior, call patterns, and memory-access regularity) provide predictive signal about which transformations are likely to be beneficial. SACT operationalizes this hypothesis by integrating structural feature extraction, adaptive relevance estimation, and stagnation-aware exploration.

We make the following contributions:

(1) **A feature-driven tuning framework** that unifies program-specific feature extraction with online relevance updates to adapt the active flag set during tuning (see Section 4.2).
(2) **An integrated structure-guided search algorithm** that combines structural analysis with adaptive surrogate modeling to reduce stagnation while maintaining sample efficiency (see Section 4.3).
(3) **An empirical evaluation** on GCC 15.2.0 using PolyBench and MiBench, comparing against BOCA, CFSCA, and `-O3`, showing improved speedup and reduced tuning cost. SACT achieves 6–9% higher speedup than both BOCA and CFSCA on the geometric mean, while requiring up to 50% fewer compile-execute measurements than BOCA (see Section 5).

2 Motivation

Compiler optimization affects performance, energy consumption, and resource utilization across domains ranging from high-performance computing to embedded systems. Yet effective flag combinations are difficult to identify due to the exponential configuration space

and the strong dependence of optimization profitability on program characteristics. Default levels such as `-O3` provide a robust baseline, but they cannot capture program-specific interactions and therefore leave performance potential unexploited.

In BOCA, the surrogate is learned solely from compile–execute feedback and ignores program structure, which can promote early over-exploitation. CFSCA uses structure only to preselect a fixed critical-flag subset, which can over-prune and cannot adapt when optimization behavior is program-specific or surrogate estimates are uncertain. The resulting challenge is to exploit structural cues during tuning while retaining an adaptively sized search space.

Figure 1 illustrates the consequence: both BOCA and CFSCA often make rapid initial progress and then plateau. For BOCA, plateauing is consistent with overconfident surrogates repeatedly sampling near an early incumbent, whereas for CFSCA it can reflect constraints imposed by the initial critical-flag subset. Such stagnation is consequential: even a 10–20% gap to a better configuration can correspond to substantial wall-clock time or energy savings at scale.

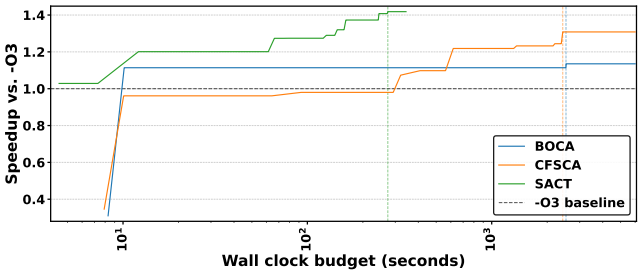


Figure 1: Optimization trajectories on PolyBench `trmm` under a 6,000 s wall-clock budget. BOCA and CFSCA evaluate configurations until the budget is exhausted, whereas SACT employs a patience-based stopping criterion and terminates after 343 s upon convergence. The vertical dashed line indicates the time at which the maximum speedup is first observed.

SACT addresses this challenge by injecting lightweight structural features into the surrogate and acquisition policy and by refining the active flag set online, reducing premature convergence without committing to a rigid static reduction.

3 Related Work

Prior work on compiler auto-tuning can be broadly grouped into (i) iterative compilation methods that explore the optimization space via repeated compile–execute measurement and (ii) learning-based approaches that replace or guide measurement with predictive models. Recent work has also emphasized standardized evaluation infrastructure (datasets, environments, and cost models) to make results comparable across optimizers and hardware. SACT draws from these directions by combining structure-informed dimensionality reduction with surrogate-guided search.

3.1 Iterative Compilation

Iterative compilation addresses flag selection by repeatedly compiling and executing a program under different flag configurations and using the resulting measurements to guide subsequent choices. Early work by Pan and Eigenmann [24] observed that a relatively small subset of flags accounts for a large fraction of attainable speedup, motivating approaches that identify relevant flags before exploring their combinations.

The literature includes a range of black-box search heuristics, including random (and stratified) sampling [9, 27], evolutionary methods such as genetic algorithms [21, 26], differential evolution [1, 33], and simulated annealing [8, 18]. These methods can be effective, but they generally require a large number of compile–execute evaluations and do not explicitly model performance for unevaluated configurations.

Bayesian optimization and related surrogate-guided methods improve sample efficiency by learning a predictive model and selecting new evaluations using an acquisition function. BOCA [9] instantiates this approach with a Random Forest surrogate (RFSM) and a decay-based acquisition strategy. More recently, systems work has proposed BO variants and scalable implementations tailored to expensive, noisy compiler objectives (e.g., BaCO [19]).

A complementary trend is improved benchmarking and reproducibility. CompilerGym [13] provides a standardized RL-style environment for compiler optimization tasks, enabling controlled comparisons between search strategies under a shared measurement pipeline.

Flag-space reduction provides another mechanism for improving efficiency. CFSCA [34] performs *a priori* dimensionality reduction by identifying *critical flags* based on static structural analysis and restricting search to the induced subspace. This reduces the effective search dimensionality, but can exclude profitable configurations when interactions span beyond the retained set. SACT adopts critical-flag selection as an initial prior, but augments it with online refinement to mitigate the rigidity of a one-shot filter.

3.2 Machine Learning for Compiler Optimization

Ashouri et al. [3] survey machine-learning approaches to compiler optimization along axes including learning objective, program representation, and evaluation methodology. Below, we highlight more recent themes most closely related to SACT.

Learning-guided heuristics in production compilers. In addition to research prototypes, recent efforts have integrated ML models into mainstream toolchains. For example, MLGO uses learned models to guide LLVM inlining and other decisions, demonstrating the practicality (and engineering constraints) of deploying ML-guided optimization policies [28].

Supervised configuration prediction. A common approach is to predict effective optimization settings directly from static program features, thereby reducing or eliminating iterative search. For example, MILEPOST GCC [15] extracts feature vectors from LLVM IR and performs nearest-neighbor transfer from a database of previously profiled programs. More recent systems use multi-stage predictors that filter and then rank candidate configurations (e.g., multiple-phase learning [35]) or learn improved heuristics by identifying informative features from iterative-compilation data (e.g., PreTuner [32]). Compared with these methods, SACT is explicitly online: it maintains a growing set of target-program measurements

and adapts search focus as evidence accumulates rather than committing to a single predicted configuration.

Surrogate-assisted optimization and transfer. Beyond BOCA, active-learning formulations use probabilistic surrogates (often Gaussian processes) to select informative evaluations that trade off exploration, exploitation, and measurement cost [6]. Related work also considers transfer across tasks and platforms, such as cross-feature transfer for tensor-program auto-tuning on new hardware [30]. SACT targets general-purpose GCC flag tuning in a single-program, online setting and leverages program structure primarily to improve search allocation rather than to perform explicit cross-program transfer.

Learned program representations. Richer program representations have been used to improve generalization across workloads, including compiler-derived graph and IR representations (e.g., control-/data-flow and data-flow graphs), token- or instruction-sequence encodings, and transformer-style models trained on code [4, 7, 11, 14, 29]. SACT instead emphasizes lightweight static features to keep feature extraction and surrogate updates inexpensive during tuning.

LLM-based optimization assistants. More recently, large language models have been explored as tools for reasoning about optimization choices and proposing candidate transformations or search policies (e.g., using LLMs to suggest optimization-flag configurations under online feedback) [12]. While these approaches differ in their core mechanism, they reinforce the value of combining static program cues with online feedback—a combination SACT operationalizes via structure-aware priors and surrogate-guided refinement.

Reinforcement learning. Reinforcement learning has also been applied to optimization selection by treating pass/flag choice as a sequential decision process [2, 20]. While such methods differ in formulation from SACT's surrogate-guided search, they share the underlying goal of improving decisions using accumulated experience.

4 SACT: Structure-Aware Compiler Auto-Tuning

SACT is a compiler auto-tuning system that integrates static program characterization with surrogate-guided search to reduce stagnation in iterative flag selection.

4.1 Problem Formulation

Let $O = \{o_1, o_2, \ldots, o_m\}$ be the set of binary optimization flags supported by the compiler. We call a particular enabled/disabled assignment of these flags an *optimization-flag configuration* (equivalently, the subset of O that is enabled). We represent a configuration by a binary vector $S = [s_1, \ldots, s_m] \in \{0,1\}^m$, yielding a search space of $2^{|O|}$ candidates. Given a program P and a compiler, let $f(P,S)$ denote the wall-clock execution time of P compiled with configuration S. The standard program auto-tuning problem is

$$S^* = \arg\min_{S \in \{0,1\}^m} f(P,S). \tag{1}$$

4.2 Feature-Driven Tuning Framework

This subsection describes the components responsible for extracting structural information from the target program and using it to reduce the optimization search space before the iterative search begins. Together, these components form a feature-driven tuning framework that adapts the active flag set to the structural characteristics of each program.

Program Feature Extraction. The feature extractor scans the source code of P statically and produces a feature vector $\phi(P) \in \mathbb{R}^d$. The feature vector is organized into four groups:

(1) **Loop features:** counts of `for`, `while`, and `do-while` loops; maximum loop-nesting depth; an indicator for nested loops; and the total number of statements in loop bodies. These features are intended to capture the potential relevance of loop optimizations such as `-funroll-loops`, `-fvectorize`, and `-floop-interchange`.
(2) **Branch features:** counts and density (branches per statement) of `if`/`else`, `switch`, and ternary constructs, which relate to flags such as `-fbranch-probabilities` and `-fprofile-arcs`.
(3) **Memory-access features:** counts of pointer dereferences, array accesses, `malloc`/`new` calls, and struct/union field accesses, which relate to flags such as `-fstrict-aliasing` and `-fomit-frame-pointer`.
(4) **Function and call-graph features:** the number of user-defined functions, average function length, estimated call depth, a recursion indicator, and total call sites, which relate to interprocedural and inlining flags such as `-fipa-cp` and `-finline-functions`.

While the individual feature families are simple, they are consistent with established practice in ML-guided compiler optimization, where lightweight static descriptors are used to characterize code structure and enable either cross-program generalization or search-space shaping (e.g., MILEPOST/iterative-compilation features [15], AutoPhase-style characterization [2], and recent learning-guided tuners [32]). Our selection process followed a pragmatic, analysis-driven criterion: we retained only (i) features that can be extracted statically with negligible overhead, (ii) features that map directly to well-understood optimization categories in GCC (loop, branch, memory, and interprocedural transformations), and (iii) features that remain stable across minor source-level changes, thereby providing a robust prior for the initial flag reduction.

We define the program feature vector as

$$\phi(P) = \left[\phi_{\text{loop}}(P), \phi_{\text{branch}}(P), \phi_{\text{mem}}(P), \phi_{\text{call}}(P)\right] \in \mathbb{R}^d \tag{2}$$

where the bracketed terms are concatenated feature subvectors and d denotes the dimension of the resulting feature vector.

SACT extracts these features once before tuning through a single-pass lexical scan of the source code, using one regular expression per syntactic construct (Algorithm 1 Line 1). Patterns identify constructs such as `while` and `for` loops, pointer dereferences, and call sites. Raw counts are normalized by the total line count of P to map each component to $[0,1]$ and reduce sensitivity to program size.

Structure-Based Flag Selection. SACT leverages CFSCA's optimization-classification table, which associates each GCC flag with the program

structures it targets (e.g., `-fvectorize` with inner loops dominated by arithmetic operations). After detecting which structures appear in P, the union of the corresponding flag sets defines O^{cfsca}. This step, denoted CFSCAFlagSelection(P, O) in Algorithm 1, provides a structurally motivated initial reduction of the flag space.

Program Embedding Initialization. The static feature vector $\phi(P)$ captures source-level structure, but the embedding $\mathbf{e}_P$ is intended to also represent performance-relevant properties that may not be fully explained by static counts. SACT initializes $\mathbf{e}_P$ from $\phi(P)$ using a learned linear projection:

$$\mathbf{e}_P^{(0)} = W_\phi\, \phi(P) + \mathbf{b}, \tag{3}$$

where $W_\phi \in \mathbb{R}^{h \times d}$ and $\mathbf{b} \in \mathbb{R}^h$ are shared parameters learned jointly with the RFSM by minimizing prediction error on the training corpus, and the embedding dimension is $h = 8$. For each program P, the projection initializes the corresponding embedding, which is refined during the search (Section 4.3). Intuitively, the projection maps static structural features into an embedding space in which proximity reflects similarity in optimization behavior.

Surrogate-Based Relevance Estimation. Given the Random Forest surrogate $\hat{f}_\theta$, whose learned parameters are denoted by θ, and the initial embedding $\mathbf{e}^{(0)}$, SACT assigns each flag $o_j \in O$ a model-based relevance score:

$$R(o_j, P) = \mathbb{E}_{\mathbf{x} \sim \mathcal{U}}\left[\left|\hat{f}_\theta(\mathbf{x}^{(o_j=1)}, \mathbf{e}^{(0)}) - \hat{f}_\theta(\mathbf{x}^{(o_j=0)}, \mathbf{e}^{(0)})\right|\right], \tag{4}$$

where $\mathbf{x}$ is a binary flag configuration drawn from the random-context distribution $\mathcal{U}$. This surrogate-only score estimates the expected absolute change in predicted runtime when toggling o_j. Random samples from $\mathcal{U}$ approximate the expectation without additional compilation.

Flags with $R(o_j, P) \geq \tau$ form the transfer-predicted relevant set $\hat{O}$, which is then intersected with the CFSCA-derived structural subset:

$$O = \hat{O} \cap O^{\text{cfsca}}. \tag{5}$$

To mitigate over-pruning when the surrogate is still inaccurate, SACT enforces a minimum space ratio: if $|O|/|O^{\text{cfsca}}| < \beta_{\min}$, the intersection is discarded and the system reverts to $O \leftarrow O^{\text{cfsca}}$.

4.3 Structure-Guided Search Algorithm

This subsection presents the iterative search algorithm that constitutes the core of SACT. The algorithm is organized into two phases. *Phase 1* extracts the static feature vector from the source code of the target program and applies the feature-driven framework described in Section 4.2 to obtain a reduced flag space O. It also initializes the program embedding $\mathbf{e}^{(0)}$ via the learned linear projection of $\phi(P)$, estimates flag relevance using surrogate sensitivity, and generates warm-start configurations by random sampling over the reduced space. *Phase 2* performs an embedding-guided Bayesian search using a surrogate model over the reduced flag space, seeded with the warm-start observations. The surrogate is retrained at each iteration, the embedding is refined online, and candidate configurations are generated by combining exhaustive enumeration of the most impactful flags with controlled random sampling of the remaining flags. Algorithm 1 summarizes the SACT procedure.

Relative to a direct composition of BOCA and CFSCA, SACT introduces two safeguards against stagnation: (i) a program embedding that is refined online and (ii) an anti-over-pruning constraint that prevents the reduced flag space from becoming excessively small when surrogate estimates are uncertain. Both mechanisms are described in Section 4.2.

Algorithm 1: SACT: Structure-Aware Compiler Auto-Tuning

Input: Program P; flag set O;
Output: Best flag configuration S^* and runtime $f(P, S^*)$

```
// Phase 1
φ ← ExtractFeatures(P);
O^cfsca ← CFSCAFlagSelection(P, O);
e^(0) = W_φ φ(P) + b; // Init embeddings
O ← RelevantFlags(f̂_θ, e, τ) ∩ O^cfsca;
if |O|/|O^cfsca| < β_min then
    O ← O^cfsca;
C_0 ← RandomSample(O, n_0);
T ← {(x, f(P, x)) | x ∈ C_0};
S* ← arg min_{x∈C_0} f(P, x);
// Phase 2
i ← 1;
while ¬Converged(S*, w) do
    if current_time ≥ time_seconds then
        break;
    model ← TrainSurrogate(T, e);
    e ← RefineEmbedding(model, T);
    foreach o_j ∈ O do
        imp[j] ← GiniImportance(o_j, model);
    O^imp ← TopK(imp, O);
    O^less ← O \ O^imp;
    B^imp ← AllSettings(O^imp);
    C^(i) ← NormalDecay(i);
    C ← [ ];
    foreach B ∈ B^imp do
        U ← RandomSettings(O^less, C^(i));
        foreach u ∈ U do
            C.add(B‖u);
    foreach S ∈ C do
        T ← T ∪ {(S, f(P, S))};
        if f(P, S) < f(P, S*) then
            S* ← S
    i ← i + 1;
return S*, f(P, S*);
```

Surrogate Training. Given the observations in $\mathcal{T}$, SACT trains a RFSM that takes as input a pair $(\mathbf{x}, \mathbf{e}_P)$ and predicts the runtime $f(P, \mathbf{x})$:

$$\hat{f}_\theta : \{0,1\}^{|O_t|} \times \mathbb{R}^h \longrightarrow \mathbb{R}, \qquad \hat{f}_\theta(\mathbf{x}, \mathbf{e}_P) \approx f(P, \mathbf{x}). \tag{6}$$

The embedding $\mathbf{e}_P$ is concatenated with the flag vector at each tree split, enabling the forest to model flag–program interaction effects (i.e., a flag can be beneficial for some programs and detrimental for others). The surrogate is implemented as an ensemble of decision trees trained on bootstrap samples of $\mathcal{T}$, appending the current embedding $\mathbf{e}$ to each flag vector so that splits can capture flag–program interaction effects. At each split, a random feature subset is considered to reduce inter-tree correlation. For a candidate S, the ensemble prediction uses the mean μ_S of tree outputs; the empirical standard deviation σ_S across trees is used as a heuristic uncertainty proxy.

Online Embedding Refinement. After each surrogate training step, SACT refines the program embedding by minimizing prediction error on the accumulated observations:

$$\mathbf{e}_t \leftarrow \arg\min_{\mathbf{e}} \sum_{(\mathbf{x},r)\in\mathcal{T}} \left(\hat{f}_\theta(\mathbf{x},\mathbf{e}) - r\right)^2, \tag{7}$$

keeping θ fixed and updating only $\mathbf{e}$. Since $\mathbf{e}$ has dimension h, this inner optimization is inexpensive (e.g., limited-memory Broyden–Fletcher–Goldfarb–Shanno (L-BFGS) or coordinate descent) relative to a compile–execute evaluation and progressively improves surrogate accuracy for P. This online adaptation mechanism specializes the embedding to the target program as measurements accumulate.

Iterative Candidate Generation. At each iteration of Phase 2, SACT identifies the most impactful flags and generates candidate configurations by combining exhaustive enumeration of those flags with controlled random sampling of the remaining ones.

First, for each flag $o_j \in O$, the normalized impurity decrease (Gini importance) attributable to o_j is computed. For a node d that splits on o_j:

$$G(d) = \left[I(d) - \omega_\ell\, I(\ell) - \omega_r\, I(r)\right] \cdot \frac{n_d}{N}, \tag{8}$$

where $I(d) = 1 - \sum_c p_c^2$ is the Gini impurity of node d, $\omega_{\ell/r} = n_{\ell/r}/n_d$, and N is the total number of training samples. Importance is aggregated across the forest as:

$$\mathrm{imp}[j] = \frac{\sum_{i=1}^{u} G(d_i)}{t}, \tag{9}$$

where u is the number of trees that split on o_j and t is the total number of trees.

The flags are sorted by descending importance and the top-K set O^{imp} is selected. The default K=8 yields an exhaustive grid of 2^8=256 settings; larger values increase cost and may exceed the time budget. All 2^K binary settings for the impactful flags are enumerated (AllSettings), ensuring complete coverage of combinations within O^{imp}.

For the less-impactful flags $O^{\mathrm{less}} = O \setminus O^{\mathrm{imp}}$, the number of random settings sampled at iteration i is controlled by a Gaussian decay schedule:

$$C(i) = C_1 \cdot \exp\left(-\frac{\max(0,\ i - \mathit{offset})^2}{2\sigma^2}\right), \quad \sigma^2 = -\frac{\mathit{scale}^2}{2\log(\mathit{decay})}. \tag{10}$$

For $i \le \mathit{offset}$, $C(i) = C_1$ (full exploration). For $i > \mathit{offset}$, $C(i)$ decreases according to the right half of a Gaussian, gradually shifting sampling from exploration to exploitation as the RFSM becomes more informative. The $C(i)$ distinct binary assignments for the less-impactful flags are sampled without replacement, avoiding re-evaluation of identical candidates within the same iteration.

Each candidate is formed by concatenating one impactful-flag setting B with one less-impactful-flag assignment u, yielding the full configuration $B\|u$. All candidates are evaluated, and the incumbent S^* is updated whenever a faster configuration is found.

Convergence. The search terminates when the incumbent best configuration S^* has not improved for w consecutive iterations, or when the time budget is exhausted.

5 Evaluation

We evaluate SACT along three axes: (i) final speedup quality relative to BOCA [9] and CFSCA [34], (ii) the number of iterations required to reach the best observed configuration, and (iii) the number of compile–execute evaluations required to reach the best observed configuration.

5.1 Methodology

Benchmark suite. We evaluate SACT on a combined benchmark suite comprising PolyBench [25] and MiBench [17]. PolyBench includes 30 compute-intensive kernels spanning linear algebra, stencil computation, and dynamic programming, which are typically sensitive to loop-level optimization flags; for each kernel we use the LARGE problem size to ensure non-trivial runtimes. MiBench adds 16 C programs from embedded-oriented domains (e.g., automotive, multimedia, networking, security, and telecommunications), using the largest provided input set when applicable. In total, the suite contains 46 programs with heterogeneous control-flow and memory-access characteristics. All programs are compiled with GCC 15.2.0. Execution time is reported as the median of five runs to mitigate noise due to OS scheduling.

Implementation. We release the full implementation of SACT and use the authors' reference implementations for BOCA[1] and CFSCA[2].

Hyperparameters. Table 1 lists the hyperparameters used for SACT. Where applicable, we adopt the defaults of BOCA and CFSCA. Parameters introduced by SACT (flag-relevance threshold τ, embedding dimension h, warm-start budget n_0, and patience window w) are selected via leave-one-out cross-validation on five held-out programs from the combined suite, stratified to include both PolyBench and MiBench workloads.

Baselines. We compare SACT against the following baselines:

- BOCA [9]: Bayesian optimization with a Random Forest surrogate, initialized with two random observations.
- CFSCA [34]: critical-flag selection followed by search in the reduced subspace.
- -O3: the GCC standard aggressive optimization level, used as the reference baseline (speedup = 1.0).

[1] https://github.com/BOCA313/BOCA
[2] https://github.com/zhumxxx/CFSCA

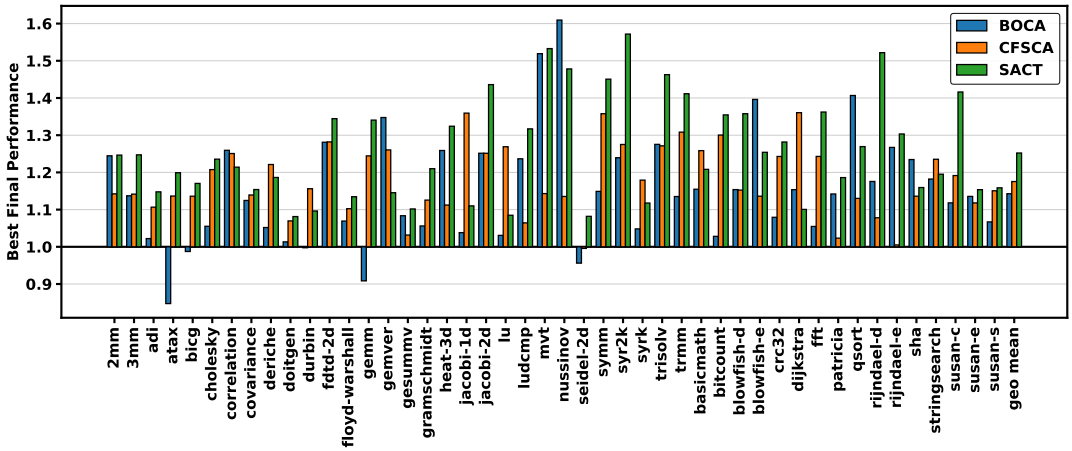


Figure 2: Speedup quality across the benchmark suite: for each benchmark (x-axis), we plot the best speedup over `-O3` achieved within the 6,000-second budget (y-axis; higher is better) for `SACT`, `BOCA`, and `CFSCA`. Dashed horizontal lines show the corresponding geometric mean speedup across all benchmarks.

Table 1: Hyperparameters used in all `SACT` experiments.

Parameter	Value	Description
τ	0.75	Flag-relevance threshold
h	8	Embedding dimension ($e_P \in \mathbb{R}^h$)
n_0	2	Warm-start budget
K	8	Top impactful flags
$\beta_{\min}$	0.3	Minimum flag-space ratio before fallback
offset	20	NormalDecay onset iteration
scale	10	NormalDecay width
decay	0.5	NormalDecay half-life factor
w	150	Patience window for Converged
iter_seconds	6000	Maximum search time
RF trees	100	Random Forest ensemble size

Compiler flag space. Following `BOCA` [9] and `CFSCA` [34], we consider m=58 binary optimization flags in `GCC`. The resulting configuration space contains 2^{58} candidate configurations, making exhaustive evaluation infeasible.

Stopping criterion. Following `CFSCA`, we use a fixed wall-clock tuning budget of 6,000 seconds.

Metrics. We report three metrics:

(1) **Speedup over `-O3`:** $\text{Speedup}(P) = T_{\text{-O3}}(P)/f(P, S^*)$, where $T_{\text{-O3}}(P)$ is the execution time under the standard `-O3` flag set and $f(P, S^*)$ is the best execution time observed during tuning. Values greater than 1 indicate improvement over `-O3`.
(2) **Evaluations:** the number of compile–execute evaluations (lower is better).
(3) **Iterations:** the number of algorithmic iterations (lower is better).

Platform. We conducted the experiments on an AMD Ryzen Threadripper PRO 7975WX 32-core system with 256 GB of RAM, running Ubuntu 24.04.4 LTS.

5.2 Speedup Quality

We first evaluate the *final tuning outcome*, i.e., the performance improvement achieved within a fixed time budget. For each of the 46 benchmarks, Figure 2 reports the best speedup over `-O3` obtained by each strategy within a 6,000-second budget. The geometric-mean speedups are 1.244× for `SACT`, 1.176× for `CFSCA`, and 1.138× for `BOCA`, with per-benchmark outcomes ranging from below 1.0× to above 1.57×. These aggregates mask qualitative differences in how the strategies succeed or fail.

Overall Ordering and Its Cause. The geometric-mean ordering (`SACT` > `CFSCA` > `BOCA`) reflects systematic differences in search guidance. `BOCA` is a purely black-box method: its Random Forest surrogate is trained only on runtime measurements from the target program and is therefore agnostic to which flags are likely to be relevant. This design permits exploration over the full 2^{58} - configuration space, but it also increases the risk of premature convergence: once the surrogate is fit to a locally concentrated set of observations, the acquisition function tends to repeatedly query the same region and progress slows down. `CFSCA` mitigates this behavior by restricting the search space *a priori* via static structural analysis, so that search is concentrated on flags associated with observed program constructs. On average, this restriction improves efficiency, but it also imposes an exclusion constraint: any high-quality configuration requiring a flag outside the selected set cannot be reached. `SACT` combines the same structural reduction with a dynamic, surrogate-driven relevance mechanism that updates the active flag set as evaluations accumulate, reducing the likelihood of becoming trapped by an overly rigid initial filter.

Where Structural Guidance Matters Most. The improvements of `SACT` over its baselines occur on benchmarks with regular, loop-dominated structure (e.g., syr2k, symm, mvt, trisolv, trmm, and jacobi-2d). In this regime, structural attributes such as loop-nest

depth, access regularity, and arithmetic intensity provide informative cues about beneficial optimizations (e.g., vectorization, unrolling, and software pipelining). By encoding these attributes, SACT can guide exploration toward a relevant subspace earlier in the budget. In contrast, BOCA lacks an explicit structural prior and consequently expends more evaluations on configurations that are unlikely to align with the program's structure. CFSCA typically selects an appropriate flag subset, but its static reduction provides limited leverage once the search reaches a local plateau.

The Challenge of Irregular and Branch-Heavy Programs. For programs with irregular control flow and input-dependent behavior, static structural features extracted from source code are often weak predictors of runtime sensitivity to specific flags. On benchmarks such as atax, gemm, bicg, and durbin, BOCA yields speedups below 1.0× in several cases, indicating that the best configuration found within budget is slower than `-O3`. This outcome is consistent with a surrogate model converging to a suboptimal region without any structure-based corrective signal. Both CFSCA and SACT avoid these severe regressions, suggesting that structural filtering can provide a useful safeguard. In this setting, the primary advantage of SACT is expressed more strongly in upper-tail improvements than in regression avoidance.

The Asymmetry Between PolyBench and MiBench. The relative advantage of SACT differs systematically across suites. PolyBench kernels exhibit dense loop structure and high arithmetic intensity, which makes them more amenable to structure-driven flag selection; correspondingly, SACT tends to outperform its baselines more consistently on PolyBench. MiBench workloads, by contrast, frequently involve irregular memory access, control-heavy execution, or intentionally data-dependent computation (e.g., patricia, dijkstra, sha), which limits the predictive value of static structure. For these programs, all methods tend to achieve smaller and closer speedups. This distinction motivates reporting suite-level breakdowns when characterizing the regime of applicability.

Why No Single Strategy Dominates Every Benchmark. Several benchmarks show that CFSCA can match SACT (e.g., fdtd-2d, correlation). This is expected when the statically selected flag set is sufficient to include the global optimum; under that condition, the smaller search space can yield faster convergence. Conversely, SACT incurs additional overhead from surrogate training and relevance scoring, which may not be amortized when static reduction already captures the optimal region. The fallback mechanism that reverts to the full CFSCA flag set when the surrogate-guided intersection would shrink the space below the minimum ratio $\beta_{\min}$ mitigates the risk of over-pruning and preserves search viability.

Robustness as a Scientific Property. Beyond mean performance, robustness across benchmarks is a key empirical property. BOCA exhibits the largest variance, achieving strong gains on some programs (e.g., nussinov) while producing the most pronounced regressions on others (e.g., atax). This variability is consistent with the absence of program-structure information in its guidance. SACT yields consistently positive speedups across the suite, with a worst-case outcome above 1.08×. From a deployment perspective, this stability is important because regressions on even a minority of programs may be unacceptable.

Summary. Overall, the speedup-quality results support three claims. (i) Structural awareness is important for robustness: purely black-box surrogate optimization can yield regressions on irregular programs within a fixed budget. (ii) Dynamic structural guidance improves over static reduction: refining the active search space during tuning is a primary contributor to SACT's advantage, particularly on compute-intensive, loop-heavy kernels. (iii) No method dominates universally: when static filtering already captures the optimal flag set, the incremental benefit of dynamic refinement diminishes, suggesting opportunities for tighter integration (e.g., using static analysis to warm-start the dynamic surrogate).

5.3 Search Efficiency and Budget Utilization

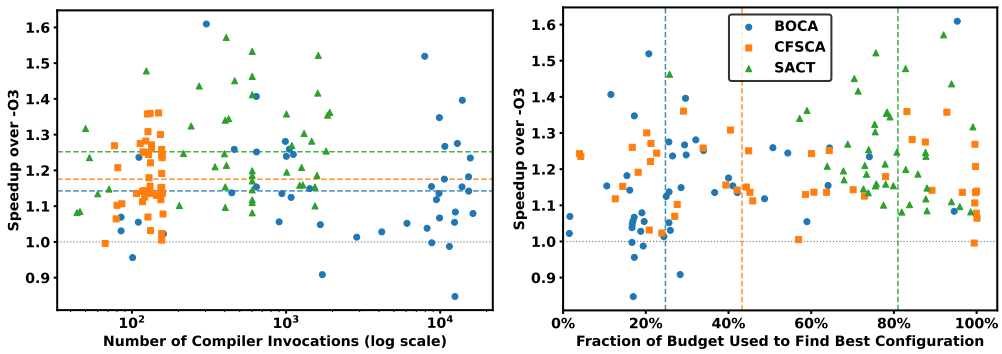
Speedup quality captures the end result of tuning. We next analyze the *search process* that produces it using (i) compile–execute evaluations completed within the budget, (ii) the fraction of wall-clock budget consumed, and (iii) the time at which the best configuration is first observed. Figure 3 reports search efficiency and budget utilization.

Evaluations and Diminishing Returns. The strategies differ not only in final speedup but also in evaluation throughput. BOCA typically issues substantially more compile–execute evaluations than CFSCA or SACT within the same 6,000-second budget, often exceeding 10,000 evaluations on benchmarks with short compile times (e.g., atax, bicg, crc32, patricia). In contrast, CFSCA evaluates a comparatively small and narrow range of configurations (approximately 80–160 per benchmark), which follows directly from its aggressive upfront flag-space reduction. SACT occupies an intermediate regime, issuing from hundreds to approximately two thousand evaluations depending on how quickly the adaptive relevance mechanism narrows the candidate set.

Evaluation count alone, however, is not predictive of speedup quality. On benchmarks where BOCA performs poorly (e.g., atax with approximately 12,500 evaluations and 0.85× speedup; bicg with approximately 11,500 evaluations and 0.99× speedup), fast compilation primarily enables denser sampling of an unproductive region of the search space. This behavior is consistent with over-exploitation: once the surrogate becomes confident in a local region, additional evaluations refine that region rather than promoting escape to alternative basins. On the same benchmarks, SACT achieves speedups above 1.19× with substantially fewer evaluations, indicating that structure-informed guidance can improve the efficiency with which the search reaches high-performing regions.

The evaluation budget of CFSCA is largely determined by its reduction step: because it searches a smaller, fixed subspace, it exhausts candidate configurations quickly and attains reasonable speedups with modest evaluation counts. The same restriction also imposes a hard reachability constraint: configurations requiring flags outside the selected subset are excluded, which can cap attainable performance even when wall-clock budget remains.

Time-to-Best Configuration. We further examine convergence dynamics using the ratio of the time at which the best configuration is found to the total budget. BOCA often identifies its best configuration early (commonly within the first 15–30% of elapsed time)



(A) Speedup vs. Search Effort (B) Speedup vs. Convergence Timing

Figure 3: Search efficiency of BOCA, CFSCA, and SACT. (A) Compiler invocations (log scale) vs. speedup over -O3. Horizontal dashed lines indicate geometric means and show whether additional evaluations improve speedup and each strategy's efficiency frontier. (B) Fraction of runtime used to find the best configuration vs. speedup. Vertical dashed lines indicate geometric means. Runtime is the 6,000 s budget for BOCA and CFSCA and SACT's actual patience-limited runtime ($\leq 2{,}000\ s$). Fractions near 0 indicate early discovery and wasted remaining budget, whereas fractions near 1 indicate late discovery.

and then stagnates. This pattern aligns with a common surrogate-search failure mode in which the acquisition function concentrates sampling in a locally optimal region, and subsequent evaluations yield limited improvement.

CFSCA exhibits the opposite pattern on several benchmarks: the best configuration is sometimes found near the end of the 6,000-second window. Because evaluations occur within a reduced flag space, successive configurations can remain qualitatively distinct late into the budget, suggesting that CFSCA may continue to make incremental progress under extended budgets; however, the maximum attainable gain remains bounded by the initial reduction.

SACT typically discovers its best configuration near the middle of its *own* run. In our experiments, the worst-case SACT run terminates by approximately 2,000 seconds. Within this shorter horizon, stagnation detection and re-exploration can still shift the best-found time later by triggering targeted exploration when improvement plateaus, reflecting continued movement into previously underexplored regions rather than repeated refinement of a single plateau.

Cost per Unit of Improvement. Taken together, these metrics characterize an efficiency trade-off between performance improvement and search cost, measured in wall-clock time and compile–execute evaluations. By prioritizing structurally plausible regions and invoking re-exploration only when needed, SACT often achieves higher speedups without saturating the full budget. This matters when tuning time is constrained by shared compute, energy, or continuous-integration latency. More broadly, the results indicate that the relationship between evaluation count and final performance should not be interpreted as monotone. Instead, performance depends primarily on how effectively evaluations are allocated across the search space. Incorporating program-structure information is one mechanism for improving that allocation.

Summary. Overall, the search-efficiency results support three claims. (i) Evaluation throughput is not a proxy for progress: BOCA often performs many more evaluations but can over-exploit a local region and stagnate. (ii) Upfront flag-space reduction improves efficiency but constrains reachability: CFSCA evaluates fewer configurations and can keep improving late in the budget, yet cannot access optima outside its selected subset. (iii) Adaptive structure-aware control yields better budget utilization: SACT typically finds strong configurations with fewer evaluations than BOCA and avoids the rigidity of purely static reduction by re-expanding and re-exploring when needed.

6 Conclusion

This paper presented SACT, a structure-aware compiler auto-tuning approach that combines static program features with surrogate-guided search to adaptively focus exploration on promising subsets of optimization flags. Across 46 benchmarks under a fixed 6,000-second budget, SACT achieved a higher geometric-mean speedup than the BOCA and CFSCA baselines while maintaining more consistent performance across programs. The results suggest that incorporating program structure improves both robustness and search efficiency. In particular, static structure can provide a useful prior that reduces unproductive evaluations, while dynamic relevance updates mitigate the rigidity of purely static flag filtering.

Future Directions. (i) The current feature representation $\phi(P)$ is purely static and source-level. Incorporating dynamic profiling data could improve embeddings for programs whose runtime behavior differs from their static structure. (ii) Finally, replacing the random-forest surrogate with a model that natively captures uncertainty, such as a Bayesian neural network or deep-kernel Gaussian process, would improve expected-improvement estimates when $\mathcal{W}_t$ is small. We plan to develop and evaluate these extensions toward a more expressive version of SACT.

Artifact Availability

The artifacts associated with this study are not publicly available at the time of publication.

References

[1] Mohamad Faiz Ahmad, Nor Ashidi Mat Isa, Wei Hong Lim, and Koon Meng Ang. 2022. Differential evolution: A recent review based on state-of-the-art works. *Alexandria Engineering Journal* 61, 5 (2022), 3831–3872. doi:10.1016/j.aej.2021.09.013

[2] Mohammed Almakki, Ayman Izzeldin, Qijing Huang, Ameer Haj Ali, and Chris Cummins. 2022. Autophase V2: Towards Function Level Phase Ordering Optimization. MLArchSys 2022. https://openreview.net/forum?id=HEWao_L2IP_

[3] Amir H. Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. 2018. A Survey on Compiler Autotuning Using Machine Learning. *Comput. Surveys* 51, 5 (2018), 96:1–96:42. doi:10.1145/3197978

[4] Amir H. Ashouri, Muhammad Asif Manzoor, Minh Vu, Raymond Zhang, Ziwen Wang, Angel Zhang, Bryan Chan, Tomasz S. Czajkowski, and Yaoqing Gao. 2024. Work-in-Progress: ACPO: An AI-Enabled Compiler Framework. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. IEEE, Raleigh, NC, USA, 21–21. doi:10.1109/CASES60062.2024.00011

[5] Amir H. Ashouri, Giovanni Mariani, Gianluca Palermo, and Cristina Silvano. 2014. A Bayesian Network Approach for Compiler Auto-Tuning for Embedded Processors. In *Proceedings of the IEEE Symposium on Embedded Systems for Real-Time Multimedia*. IEEE, Greater Noida, India, 90–97. doi:10.1109/ESTIMedia.2014.6962349

[6] Prasanna Balaprakash, Robert B. Gramacy, and Stefan M. Wild. 2013. Active-learning-based surrogate models for empirical performance tuning. In *Proceedings of the International Conference on Cluster Computing*. IEEE, Indianapolis, IN, USA, 1–8. doi:10.1109/CLUSTER.2013.6702683

[7] Alexander Brauckmann, Andrés Goens, Sebastian Ertel, and Jeronimo Castrillon. 2020. Compiler-based graph representations for deep learning models of code. In *Proceedings of the International Conference on Compiler Construction* (San Diego, CA, USA) *(CC 2020)*. Association for Computing Machinery, New York, NY, USA, 201–211. doi:10.1145/3377555.3377894

[8] Martin Chak, Nikolas Kantas, and Grigorios A Pavliotis. 2023. On the Generalized Langevin Equation for Simulated Annealing. *SIAM/ASA Journal on Uncertainty Quantification* 11, 1 (2023), 139–167. doi:10.1137/21M1462970

[9] Junjie Chen, Ningxin Xu, Peiqi Chen, and Hongyu Zhang. 2021. Efficient Compiler Autotuning via Bayesian Optimization. In *Proceedings of the International Conference on Software Engineering*. IEEE, Spain, 1198–1209. doi:10.1109/ICSE43902.2021.00110

[10] Yang Chen, Shuangde Fang, Yuanjie Huang, Lieven Eeckhout, Grigori Fursin, Olivier Temam, and Chengyong Wu. 2012. Deconstructing Iterative Optimization. *ACM Transactions on Architecture and Code Optimization (TACO)* 9, 3 (2012), 21:1–21:30. doi:10.1145/2355585.2355594

[11] Chris Cummins, Zacharias V. Fisches, Tal Ben-Nun, Torsten Hoefler, Michael F. P. O'Boyle, and Hugh Leather. 2021. ProGraML: A Graph-based Program Representation for Data Flow Analysis and Compiler Optimizations. In *Proceedings of the International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR, Virtual Event, 2244–2253. https://proceedings.mlr.press/v139/cummins21a.html

[12] Chris Cummins, Volker Seeker, Dejan Grubisic, Mostafa Elhoushi, Youwei Liang, Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Kim Hazelwood, Gabriel Synnaeve, and Hugh Leather. 2023. Large Language Models for Compiler Optimization. arXiv:2309.07062 [cs.PL] https://arxiv.org/abs/2309.07062

[13] Chris Cummins, Bram Wasti, Jiadong Guo, Brandon Cui, Jason Ansel, Sahir Gomez, Somya Jain, Jia Liu, Olivier Teytaud, Benoit Steiner, Yuandong Tian, and Hugh Leather. 2022. CompilerGym: robust, performant compiler optimization environments for AI research. In *Proceedings of the International Symposium on Code Generation and Optimization* (Virtual Event, Republic of Korea) *(CGO '22)*. IEEE Press, Seoul, Republic of Korea, 92–105. doi:10.1109/CGO53902.2022.9741258

[14] Chaoyi Deng, Jialong Wu, Ningya Feng, Jianmin Wang, and Mingsheng Long. 2025. CompilerDream: Learning a Compiler World Model for General Code Optimization. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.2*. Association for Computing Machinery, Toronto, ON, Canada, 486–497. doi:10.1145/3711896.3736887

[15] Grigori Fursin, Yuriy Kashnikov, Abdul Wahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Bilha Mendelson, Ayal Zaks, Eric Courtois, François Bodin, Phil Barnard, Elton Ashton, Edwin Bonilla, John Thomson, Christopher Williams, and Michael O'Boyle. 2011. Milepost GCC: Machine Learning Enabled Self-tuning Compiler. *International Journal of Parallel Programming* 39, 3 (2011), 296–327. doi:10.1007/s10766-010-0161-2

[16] Unai Garciarena and Roberto Santana. 2016. Evolutionary Optimization of Compiler Flag Selection by Learning and Exploiting Flags Interactions. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion* (Denver, Colorado, USA). Association for Computing Machinery, New York, NY, USA, 1159–1166. doi:10.1145/2908961.2931696

[17] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the International Workshop on Workload Characterization*. IEEE Computer Society, USA, 3–14. doi:10.1109/WWC.2001.990739

[18] Mohd. Sayemul Haque, Md. Fahim, and Muhammad Ibrahim. 2023. An Exploratory Study on Simulated Annealing for Feature Selection in Learning-to-Rank. arXiv:2310.13269 [cs.LG] https://arxiv.org/abs/2310.13269

[19] Erik Orm Hellsten, Artur Souza, Johannes Lenfers, Rubens Lacouture, Olivia Hsu, Adel Ejjeh, Fredrik Kjolstad, Michel Steuwer, Kunle Olukotun, and Luigi Nardi. 2023. BaCO: A Fast and Portable Bayesian Compiler Optimization Framework. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (Vancouver, BC, Canada) *(ASPLOS '23)*. Association for Computing Machinery, New York, NY, USA, 19–42. doi:10.1145/3623278.3624770

[20] Shalini Jain, Yashas Andaluri, S. VenkataKeerthy, and Ramakrishna Upadrasta. 2022. POSET-RL: Phase Ordering for Optimizing Size and Execution Time Using Reinforcement Learning. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE, Singapore, Singapore, 121–131. doi:10.1109/ISPASS55109.2022.00012

[21] Jian-Yu Li, Zhi-Hui Zhan, and Jun Zhang. 2022. Evolutionary computation for expensive optimization: A survey. *Machine Intelligence Research* 19, 1 (2022), 3–23. doi:10.1007/s11633-022-1317-4

[22] San-Chih Lin, Chi-Kuang Chang, and Nai-Wei Lin. 2008. Automatic selection of GCC optimization options using a gene weighted genetic algorithm. In *Proceedings of the Asia-Pacific Computer Systems Architecture Conference*. IEEE, Hsinchu, Taiwan, 1–8. doi:10.1109/APCSAC.2008.4625477

[23] Vivek Nair, Zhe Yu, Tim Menzies, Norbert Siegmund, and Sven Apel. 2020. Finding Faster Configurations Using FLASH. *IEEE Transactions on Software Engineering* 46, 7 (2020), 794–811. doi:10.1109/TSE.2018.2870895

[24] Zhelong Pan and Rudolf Eigenmann. 2006. Fast and Effective Orchestration of Compiler Optimizations for Automatic Performance Tuning. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '06)*. IEEE Computer Society, USA, 319–332. doi:10.1109/CGO.2006.38

[25] Louis-Noël Pouchet. 2012. PolyBench/C: The Polyhedral Benchmark Suite. Software benchmark suite, version 3.2. https://www.cs.colostate.edu/~pouchet/software/polybench/polybench.html

[26] Anderson Faustino da Silva, Bernardo N. B. de Lima, and Fernando Magno Quintão Pereira. 2021. Exploring the space of optimization sequences for code-size reduction: insights and tools. In *Proceedings of the International Conference on Compiler Construction* (Virtual, Republic of Korea) *(CC 2021)*. Association for Computing Machinery, New York, NY, USA, 47–58. doi:10.1145/3446804.3446849

[27] Jacob O. Tørring, Carl Hvarfner, Luigi Nardi, and Magnus Själander. 2025. CAT-Bench: A Compiler Autotuning Benchmarking Suite for Black-box Optimization. In *Proceedings of the Fourth International Conference on Automated Machine Learning (Proceedings of Machine Learning Research, Vol. 293)*, Leman Akoglu, Carola Doerr, Jan N. van Rijn, Roman Garnett, and Jacob R. Gardner (Eds.). PMLR, New York, NY, USA, 24/1–20. https://proceedings.mlr.press/v293/torring25a.html

[28] Mircea Trofin, Yundi Qian, Eugene Brevdo, Zinan Lin, Krzysztof Choromanski, and David Li. 2021. MLGO: A Machine Learning Guided Compiler Optimizations Framework. arXiv:2101.04808 [cs.PL] https://arxiv.org/abs/2101.04808

[29] S. VenkataKeerthy, Rohit Aggarwal, Shalini Jain, Maunendra Sankar Desarkar, Ramakrishna Upadrasta, and Y. N. Srikant. 2020. IR2VEC: LLVM IR Based Scalable Program Embeddings. *ACM Trans. Archit. Code Optim.* 17, 4, Article 32 (Dec. 2020), 27 pages. doi:10.1145/3418463

[30] Gaurav Verma, Siddhisanket Raskar, Murali Emani, and Barbara Chapman. 2024. Cross-Feature Transfer Learning for Efficient Tensor Program Generation. *Applied Sciences* 14, 2 (2024). doi:10.3390/app14020513

[31] Zheng Wang and Michael F. P. O'Boyle. 2018. Machine Learning in Compiler Optimization. *Proc. IEEE* 106, 11 (2018), 1879–1901. doi:10.1109/JPROC.2018.2817118

[32] Guojian Xiao, Haoxuan Pi, Yifan Bai, Kuan Li, and Jianping Yin. 2025. PreTuner: Compiler Flag Auto-Tuning Based on Iterative Compilation and Predictive Model. In *Proceedings of the International Conference on Computer Supported Cooperative Work in Design*, Weiming Shen, Marie-Hélène Abel, Nada Matta, Jean-Paul A. Barthès, Junzhou Luo, Jinghui Zhang, Haibin Zhu, and Kunkun Peng (Eds.). IEEE, Compiegne, France, 1998–2004. doi:10.1109/CSCWD64889.2025.11033656

[33] Haigang Zhang and Da Wang. 2022. An External Selection Mechanism for Differential Evolution Algorithm. *Computational Intelligence and Neuroscience* 2022 (2022), 1–18. doi:10.1155/2022/4544818

[34] Mingxuan Zhu and Dan Hao. 2023. Compiler Auto-Tuning via Critical Flag Selection. In *Proceedings of the International Conference on Automated Software Engineering*. IEEE Press, Echternach, Luxembourg, 1000–1011. doi:10.1109/ASE56229.2023.00209

[35] Mingxuan Zhu, Dan Hao, and Junjie Chen. 2024. Compiler Autotuning through Multiple-phase Learning. *ACM Transactions on Software Engineering and Methodology* 33, 4, Article 100 (April 2024), 38 pages. doi:10.1145/3640330