Compiler Optimization-Based SMT Simplifications: An In-Depth Study

HANYUN JIANG, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China
PEISEN YAO*, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China
JIACHEN LU, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China
YONGWANG ZHAO, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China
KUI REN, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China

SMT solvers are a cornerstone in numerous domains, such as program verification, repair, and synthesis. While formula simplification is a crucial preprocessing step that directly impacts solver efficiency, existing approaches predominantly rely on handcrafted heuristics. Recent advances have explored using compiler optimizations to automate and enhance formula simplifications. Despite their promise, the synergy between compiler optimizations and SMT simplifications remains insufficiently understood and underexploited. This paper presents a comprehensive study of the interplay between compiler optimizations and SMT formula simplifications. We show that iterative search for optimization configurations significantly improves formula simplifications, yielding a geometric mean speed-up of 2.96× over default solvers and 2.12× over SLOT on large-scale benchmarks. We present strong evidence for the effectiveness of machine learning-based methods in automating the selection of optimization configurations tailored to specific problem instances, achieving geometric mean speed-ups of 1.15× over Z3 and 1.54× over CVC5. Finally, we discuss promising directions for improving the applicability and effectiveness of compiler optimization-driven approaches to SMT simplifications.

CCS Concepts: • **Software and its engineering → Software notations and tools**; • **Computing methodologies → Machine learning**;

Additional Key Words and Phrases: SMT solving, compiler optimizations, machine learning, corpus study

ACM Reference Format:
Hanyun Jiang, Peisen Yao, Jiachen Lu, Yongwang Zhao, and Kui Ren. 2026. Compiler Optimization-Based SMT Simplifications: An In-Depth Study. 1, 1 (February 2026), 36 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Satisfiability Modulo Theories (SMT) solvers decide the satisfiability of formulas over first-order theories, such as integers, reals, bit-vectors, and strings. They have become the underlying engines of various client applications, such as symbolic execution [17, 35, 37, 76, 97], refinement type [80], predictive trace analysis [38], semantics-based program repair [55], and program synthesis [11]. Beyond academic success, SMT solvers are widely adopted in the industry for

*Corresponding author

Authors' addresses: Hanyun Jiang, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China, jhanyun@zju.edu.cn; Peisen Yao, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China, pyaoaa@zju.edu.cn; Jiachen Lu, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China, lujc@zju.edu.cn; Yongwang Zhao, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China, zhaoyw@zju.edu.cn; Kui Ren, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China, kuiren@zju.edu.cn.

Manuscript submitted to ACM

addressing software engineering tasks, such as detecting zero-day vulnerabilities in Windows [35], verifying the safety of radiotherapy machine [62], and enforcing access control policies at Amazon Web Services [13, 24].

Despite their widespread use, the success of SMT solvers remains hindered by the scalability challenges inherent in constraint solving. A pivotal factor influencing the efficiency of SMT solvers is the simplification of input formulas, typically performed by a component known as the *rewriter*, which applies a series of rewriting rules to reduce the size and complexity of the formulas. Numerous studies have underscored the importance of enhancing the simplification procedure to boost SMT solving [58, 60, 61, 63, 65, 73, 91]. As such, previous research has explored various approaches to simplifying SMT formulas, including theory-specific methods (e.g., strength reduction, bit-width reduction) and theory-agnostic techniques (e.g., disjunctive partitioning).

Unfortunately, designing and implementing effective SMT simplifications is challenging and requires deep domain expertise, extensive analysis of problem instances, and significant engineering efforts. As a remedy, several efforts have automatically synthesized rewriting rules for specific theories, such as strings [66] and bit-vectors [60, 61]. For example, Nötzli et al. [61] use Syntax-Guided Synthesis (SyGuS) to enumerate rewrite rules efficiently, and SWAPPER [73] combines machine learning and constraint-based synthesis to generate rewrite patterns. Despite these advancements, there remains substantial room for improvement in the design of SMT simplifications. Besides, current methods are mainly implemented and evaluated on specific SMT solvers and are difficult to adapt to other ones.

In parallel, the compiler optimization community has developed robust techniques for simplifying program intermediate representations (IRs). Indeed, certain SMT simplifications closely resemble compiler optimizations, such as peephole optimization, which applies localized transformations to small instruction sequences. The structural similarities between SMT-LIB inputs and compiler IRs and their respective simplification strategies offer an opportunity to enhance SMT solving by adapting well-established compiler optimizations to this domain [63, 72, 91]. More recently, Mikek and Zhang [57] introduced SLOT, a framework that leverages mature compiler optimizations to enhance SMT solving. SLOT operates by translating SMT constraints into LLVM IR, applying standard LLVM optimizations, and then converting the optimized IR back into SMT. This preprocessing step, independent of the SMT solver itself, has demonstrated considerable potential in simplifying SMT formulas, thereby reducing solving overhead.

Despite the promise shown by previous work that utilizes the synergies between compiler optimizations and SMT simplifications [57, 72, 91], there are notable limitations:

- *Lacking Analysis of the Combined Impacts of Multiple Optimizations.* Compiler optimizations often interact in nontrivial ways, where applying one optimization can affect the effectiveness of others. Existing work lacks a systematic analysis of these interactions and of how different combinations of optimizations can be orchestrated to improve SMT-solving performance.
- *Missing Opportunities of Instance-specific Simplifications.* SLOT applies uniform optimizations across all SMT instances. However, prior research demonstrates that the effectiveness of such simplifications can vary significantly based on the characteristics of the input program. By leveraging compiler-based transformations, instance-aware simplifications can exploit program-specific properties, enabling additional performance gains that uniform approaches may fail to achieve.

This paper addresses existing limitations by systematically investigating the effects of compiler optimizations on SMT solver performance. We begin by providing an overview of prominent SMT simplification techniques and major compiler optimization strategies. By examining their synergies, we highlight how prior work has leveraged these

connections and identify areas where existing methods fall short. In particular, we focus on two key research questions that we believe are important for advancing the state of the art in this area:

- **RQ1**: How do different combinations of compiler optimizations affect SMT solver performance? What potential performance improvements are achievable through different combinations of compiler optimizations on SMT solver performance, and how can we effectively exploit these synergistic optimization opportunities?
- **RQ2**: To what extent can machine learning methods effectively predict suitable optimization configurations? How do problem-specific features and the choice of machine learning models impact the accuracy and utility of such predictions?

While prior work has applied compiler optimizations in the context of symbolic execution [18, 32], these efforts operate at a different level of abstraction. Specifically, they target compiler IRs derived from full programs, aiming to address challenges such as path explosion, memory modeling, and handling library code. In contrast, our focus is on SMT formulas that have already been extracted from source programs or other analyses. These formulas originate from a wide range of domains, such as software verification and cryptographic analysis. Although they may share a common logic, their structural properties vary significantly due to differences in source language, target properties, encoding strategies, and other factors. This heterogeneity introduces distinct challenges: effective simplification must account for diverse formula structures, without relying on assumptions specific to a particular domain or problem encoding.

To address RQ1, we rigorously evaluate the impact of various compiler optimizations on two SMT solvers (Z3 and CVC5) using standard benchmarks from SMT-COMP (§ 5). Our results uncover significant variability in how different optimizations influence solver performance. Specifically, we find that the choice of optimizations can dramatically affect runtime, underscoring the need to tailor them to specific problem characteristics. For instance, using the optimal set of optimizations for a given SMT input leads to an average performance speedup of 2.12× over SLOT and 2.96× over Z3's default configuration. Our approach achieves markedly better results on complex constraints with solving times exceeding 30 seconds.

Building on these insights, to address RQ2, we further investigate how SMT simplifications can be guided by predicting compiler optimizations that yield significant performance improvements (§ 6). The prediction problem is formulated as a multi-label classification task. By combining IR2Vec-based features and the decision tree, our model predicts optimizations demonstrating improved performance over default solvers and SLOT, achieving average speedups of 1.15× and 1.54× for Z3, respectively. Our results provide strong evidence for the effectiveness of leveraging machine learning for per-instance simplifications. Additionally, we compare the impact of different machine learning algorithms and problem features. To further validate the generalizability of our TUNA-Learn predictor, we conduct a case study on a challenging, external benchmark suite drawn from QSF [93]. This dataset was not used in our training or evaluation for RQ1 and RQ2. This case study demonstrates TUNA-Learn's ability to generalize to unseen floating-point instances (QF_FP) and unknown upstream applications, achieving 1.27× speedup over default solvers. In summary, this paper makes the following key contributions:

- We extensively study the combined effects of compiler optimizations on SMT simplifications, revealing how different optimization strategies interact and influence solver performance.
- We demonstrate a data-driven approach to predicting optimization configurations for SMT formulas, which outperforms the state-of-the-art for speeding up two existing SMT solvers.
- We discuss our findings thoroughly and identify open research directions for advancing the applicability and effectiveness of compiler optimization-based SMT simplifications.

2 PRELIMINARIES

This section provides a detailed background on formula simplifications in SMT solving (§ 2.1), compiler optimizations (§ 2.2), and their connections (§ 2.3).

2.1 Simplifications in SMT Solvers

Simplifying formulas is critical to improving the performance of both eager and lazy approaches to SMT solving [58, 61]. In general, SMT simplifications fall into two main categories:

- *Rewrites*, which replace terms in a formula with other terms in the same logic that are easier to reason about. For example, one could replace a complicated term with a simpler one with the same meaning, such as constant folding (e.g., $2 + 2$ is immediately replaced with 4) and strength reduction [7] (expensive operations, like multiplication, are replaced with less costly operations, such as additions).
- *Reductions*, which reduce a problem to an equivalent problem in another logic, solvable by a more efficient or easier-to-implement solver. For example, incremental liberalization [22, 23] reduces nonlinear formulas to linear ones, word-blasting translates floating points to bit-vectors, and bit-blasting reduces bit-vector constraints to Boolean constraints.

This paper primarily focuses on the various rewrites used in simplifications. In what follows, we may use "simplifications" and "rewriting" interchangeably.

Categories of Rewriters. Rewriting can be applied to assertions iteratively until no further rewriting rules can be applied, resulting in a fixed point. SMT solvers employ a variety of rewriting strategies, which can be categorized along several dimensions:

- *General vs. Domain-Specific*. General simplifications apply across multiple theories, while domain-specific simplifications are tailored to specific logical theories. For example, interval constraint propagation is an arithmetic-specific technique, and Ackermannization eliminates uninterpreted functions. General simplifications, on the other hand, work across various first-order theories. Flattening, for example, breaks down complex logical conjunctions into simpler components, e.g., $a \wedge (b \wedge (c = d))$ can be flattened into $\{a, b, c = d\}$.
- *Local vs. Global*. Local simplifications operate on small parts of the formula, often focusing on individual terms or atoms. For instance, in the bit-vector domain, the two's complement $-x$ of a bit-vector term x can be rewritten to $(\sim x + 1)$ using one's complement and bit-vector addition. And $(t + 0)$ can be rewritten to t. Global simplifications, on the other hand, analyze the entire set of assertions and apply transformations that span multiple terms. For example, extract elimination removes bit-vector extracts over constants, and lambda elimination applies beta reduction to lambda expressions across the entire formula.

Benefits of the Rewriters. The use of rewriting rules assists SMT solving in several dimensions. First, they *normalize* the input formulas to a normal form where specific solving algorithms can operate over, such as using Tseitin transformation [45] for conjunctive normal form (CNF) and applying Skolemization to obtain the Skolem normal form (SNF). Second, they *reduce* the overhead of the underlying solving algorithms, which would otherwise struggle to handle the constraints. For example, modern string solvers rely heavily on simplifications to improve the performance of the CDCL(T) engine [65]. Third, they serve as *semi-decision procedures* that can directly decide specific formulas' (un)satisfiability. For example, the simplifications can identify trivial conflicts such as $x = 1 \wedge x = 2$. General bit-vector

Table 1. Examples of LLVM optimizations

Granularity	Typical optimization passes
Single instruction	lower-invoke, lower-atomic, lower-expect, lower-constant-intrinsics
Loop	loop-flatten, loop-interchange, loop-rotate,loop-sink
Function	newgvn, licm, adce, sccp, aggressive-instcombine, irce
Module	ipsccp, lower-ifunc, wholeprogramdevirt,wglobaldce

linear equalities can be solved using a modified Gaussian elimination [21] that takes care of modulo-2^n multiplication, addition, and equality.

For these reasons, simplifications are critical components of SMT solvers. In particular, the Spear solver reportedly has approximately 160 rewriting rules [7], and the MathSAT solver contains nearly 300 rewriting rules [33].

2.2 Compiler Optimizations

Compiler optimizations consist of sequences of program transformations that convert a program into another semantically equivalent yet (typically) more efficient one. Modern compilers have dedicated decades of research and engineering efforts to developing powerful optimizations. As exemplified in Table 1, modern compilers provide a suite of optimization passes, each enabled by specific flags (e.g., `-funroll-loops` for loop unrolling). They also support optimization levels (`-O0` to `-O3`), which represent increasing degrees of optimization. Lower levels apply lightweight transformations to reduce code size and compilation time. In comparison, higher levels (e.g., `-O3`) employ more aggressive techniques to maximize runtime performance, often at the expense of longer compilation time and larger binaries.

Compiler Auto-tuning. The vast number of available optimization passes and their intricate interactions often hinder compilers from identifying the optimal sequence of transformations. Auto-tuning has emerged as a promising approach to automatically selecting the most effective optimization sequences. To date, most efforts focus on the *iterative compilation* [3, 84] paradigm, which involves recompiling the application multiple times with different compiler optimizations. Various search algorithms have been explored, such as Bayesian optimization [19] and genetic algorithms [77]. In addition to applying optimization algorithms, some works [100] also use machine learning to guide the iterative selection process based on the program's characteristics. Despite the research progress, iterative tuning can be time-consuming, especially for complex programs, because it requires numerous recompilation runs.

2.3 Synergy between Compiler Optimizations and SMT Simplifications

What are the Synergies? Although SMT simplification and compiler optimization originate from distinct domains, they employ analogous techniques, such as constant propagation, to eliminate redundant computations. Recognizing the structural parallels between these areas enables the transfer of ideas and methods, potentially improving the effectiveness of both. We identify several key synergies between SMT solving and compiler optimization:

- First, many SMT simplifications resemble compiler optimizations. For instance, strength reduction, a technique commonly used in compilers, has been adapted to simplify bit-vector SMT formulas [7]. This technique replaces expensive operations such as multiplication and division with simpler operations, such as shifts and additions, reducing the overall complexity of the formula and speeding up the SAT-solving phase in bit-blasting-based SMT solvers.

Table 2. Comparison of pipeline stage timings (in seconds)

Methods	SMT->IR (s)	Optimize (s)	IR->SMT (s)	Solving (s)	Total (s)
Ours	0.429	0.064	0.005	0.042	0.540
SLOT	0.432	0.045	0.004	6.041	6.522
Z3	/	/	/	7.308	7.308

- Second, compiler optimizations can introduce new opportunities for SMT simplification. For example, global value numbering (GVN) [67] eliminates redundant values by reasoning about control and data dependence within a function. However, in SMT solvers, such contextual simplifications can be heavyweight, requiring several SMT calls, as in Dillig et al. [30]'s algorithm and the `ctx-simplify` tactic of Z3.
- Third, certain simplifications are unique to SMT solving and go beyond traditional compiler optimizations. For instance, unconstrained variable elimination [14] is a powerful simplification technique in SMT solvers that removes variables that do not affect the satisfiability. This technique goes beyond what compiler optimizations offer and highlights the specialized nature of SMT solving; e.g., simplification can be satisfiability-preserving rather than semantics-preserving.

How are the Synergies Utilized. Despite these parallels, SMT simplifications and compiler optimizations serve different goals. Compiler optimizations focus on runtime performance, while SMT simplifications aim to streamline formula solving. However, we have noticed several efforts to utilize the synergies between the two fields.

- *Apply IR-level Optimizations before Generating SMT Constraints.* This is the most standard approach to utilizing the synergy. In SMT-based program analysis tools such as symbolic executors and bounded model checkers, applying compiler optimizations, such as constant folding and dead code elimination, before encoding the program into SMT constraints can significantly reduce the complexity of the resulting formulas [63, 91].
- *Implement SMT Simplifications inside Compiler IRs.* Shi et al. [72] implement simplifications and bit-blaster on top of the program dependence graphs (PDGs). Their key insight is that PDGs and path conditions are allotropes, meaning that although they appear in different forms, they encode the same program information, i.e., control and data dependencies. Thus, they propose directly determining path feasibility using the IR, without explicitly computing many exponentially sized path conditions, thereby saving time and memory.
- *Translate SMT Constraints to IRs for Using Compiler Optimizations.* SLOT [57] demonstrates that compiler optimizations can be used to simplify SMT formulas *after* they are generated. This approach leverages compiler techniques by translating SMT constraints into LLVM IR, applying standard compiler optimizations, and then converting the optimized IR back into SMT formulas.

Example 2.1. As shown in Figure 1, this constraint encodes a set of constraints over four 8-bit bit-vector variables using a combination of arithmetic and bitwise operations such as zero-extension, addition, subtraction, left-shift, and bitwise-and. The constraints simulate a symbolic execution path condition extracted from dynamic test generation (e.g., SAGE), and check whether a specific non-zero difference satisfies several signed inequality bounds. The solver is asked to check the satisfiability of this formula, which is known to be `unsat`. These elements make it a representative and challenging QF_BV benchmark.

```
(set-info :smt-lib-version 2.6)
(set-logic QF_BV)
(set-info :category "industrial")
(set-info :status unsat)
(declare-fun T1_7389 () (_ BitVec 8))
(declare-fun T1_7390 () (_ BitVec 8))
(declare-fun T1_7431 () (_ BitVec 8))
(declare-fun T1_7432 () (_ BitVec 8))
(assert (let ((?v_3 ((_ zero_extend 24) (_ bv1 8)))
(?v_2 ((_ zero_extend 24) (_ bv2 8))) (?v_0 (bvsub ((_
zero_extend 24) T1_7389) (_ bv48 32)))) (let ((?v_6
(bvsub (bvadd ((_ zero_extend 24) T1_7390) (bvshl
(bvadd ?v_0 (bvshl ?v_0 ?v_2)) ?v_3)) (_ bv48 32)))
(?v_1 (bvsub ((_ zero_extend 24) T1_7431) (_ bv48 32))))
(let ((?v_5 (bvand (bvsub (bvadd ((_ zero_extend 24)
T1_7432) (bvshl (bvadd ?v_1 (bvshl ?v_1 ?v_2)) ?v_3)) (_
bv48 32)) (_ bv4294967295 32)))) (let ((?v_4
(bvsub ?v_6 ?v_5))) (and true (not (= ?v_4 (_ bv0 32)))
(bvsle (bvadd ?v_4 ?v_3) (_ bv1 32)) (bvsle ?v_5 (_
bv255 32)) (bvsle (_ bv0 32) ?v_5) (bvsle ?v_5 ?v_6)
(bvsle ?v_6 (_ bv255 32)) (bvsle (_ bv0 32) ?v_6)))))))
(check-sat)
(exit)
```

(a) Original constraint

```
(set-info :status unknown)
(declare-fun T1_7431 () (_ BitVec 8))
(declare-fun T1_7389 () (_ BitVec 8))
(declare-fun T1_7432 () (_ BitVec 8))
(declare-fun T1_7390 () (_ BitVec 8))
(assert
(let ((?x33 (bvadd (bvmul ((_ zero_extend 24) T1_7389) (_ bv5
32)) (bvmul ((_ zero_extend 24) T1_7431) (_ bv2147483643 32)))))
(let ((?x38 (bvadd (bvsub (bvadd ((_ zero_extend 24) T1_7390) (_
bv1 32)) ((_ zero_extend 24) T1_7432)) (bvshl ?x33 (_ bv1 32)))))
(let ((?x24 (bvadd (bvadd ((_ zero_extend 24) T1_7432) (_
bv4294966816 32)) (bvmul ((_ zero_extend 24) T1_7431) (_ bv10
32)))))
(let ((?x16 (bvadd (bvadd ((_ zero_extend 24) T1_7390) (_
bv4294966816 32)) (bvmul ((_ zero_extend 24) T1_7389) (_ bv10
32)))))
(let (($x42 (and (bvult (bvadd ?x24 (_ bv4294967248 32)) (_
bv256 32)) (and (and (distinct ?x16 ?x24) true) (bvslt ?x38 (_
bv2 32))))))
(and (bvult (bvadd ?x16 (_ bv4294967248 32)) (_ bv256 32)) (and
(bvsle ?x24 ?x16) $x42))))))))
(check-sat)
```

(b) Optimized constraint with SLOT

```
(set-info :status unknown)
(assert
false)
(check-sat)
```

(c) Optimized constraint with our method

Fig. 1. Compare the simplification performance of our method and SLOT. (a) is the original constraint from SMT-COMP, generated by the SAGE symbolic executor. (b) and (c) are the simplified constraints after applying SLOT and our approach, respectively.

When solved directly using Z3, the constraint requires 3.470 seconds. In contrast, our approach achieves a speedup of 12.07× over the SLOT baseline and 13.53× over Z3's default configuration. A detailed comparison is provided in Table 2, clearly demonstrating the effectiveness of our synergy-driven optimization strategy.

3 PROBLEM STATEMENT

While prior work, such as SLOT [57], has demonstrated the promise of leveraging compiler optimizations to simplify SMT formulas and improve solver performance, significant challenges remain in fully realizing these benefits. Specifically, it remains unclear which compiler optimizations are the most effective and under what conditions they should be applied. The intrinsic complexity of SMT solving, combined with the wide array of available optimizations, raises several open questions that have yet to be systematically addressed:

- First, the interactions between different optimizations are non-trivial, and their combined effects on solver performance are neither additive nor easily predictable. The current literature lacks a systematic exploration of how different combinations of compiler optimizations impact SMT solving. While auto-tuning has been successfully applied in compiler optimization, its application in our context has not been investigated.
- Second, much of the prior work on compiler auto-tuning has employed iterative search methods that repeatedly compile a program and evaluate optimization configurations. However, this strategy can be impractical for SMT solvers due to the overhead of evaluating each configuration. A more feasible alternative may be directly predicting suitable configurations. Nevertheless, instance-specific prediction presents challenges due to the vast search space and diversity of SMT problems.

Prior work has examined the interaction between compiler optimizations and symbolic execution. For example, Chen et al.[18] develop techniques to predict optimization sequences that improve symbolic execution performance. In contrast, our setting is domain-agnostic and must accommodate heterogeneous constraint structures arising from a

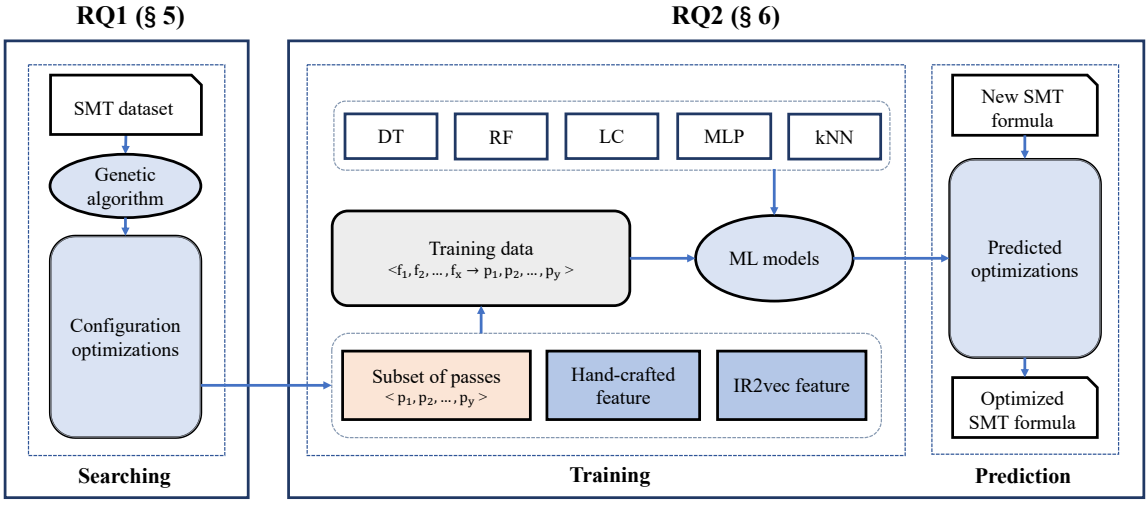


Fig. 2. Overall workflow of the evaluation methodology TUNA.

wide range of applications. However, by leveraging SLOT, we operate in a regime with distinctive structural properties: the SMT formulae correspond to single-function, loop-free IRs with no external calls. Thus, while our target domains may vary widely, the IRs we analyze exhibit a constrained and well-defined structure This structural regularity raises a natural question: *can we exploit this uniformity to develop principled methods for compiler-based SMT simplifications?*

In an attempt to address this question, our study centers on two distinct yet complementary perspectives.

(The optimization problem): Given a set of formulas F, a set of optimization passes P, and an SMT solver S, the goal is to determine an optimal subset of P that maximizes the performance of S on F, subject to resource constraints.

(The prediction problem): Given an SMT-LIB2 input and a set of optimization passes P, develop a machine learning model that predicts the most effective subset of P for improving solver performance on that input.

Problem Scope. Compiler auto-tuning encompasses two core challenges: pass selection (determining which optimization passes to enable) and phase ordering (determining the order in which passes are applied). In line with prior work [3, 19, 99, 100], we focus exclusively on pass selection. Phase ordering is excluded from consideration due to its inherent combinatorial complexity and the difficulty of modeling pass interactions. To control this complexity, we adopt a fixed pass ordering identical to that used in LLVM's `-O3` optimization level.

Our focus is restricted to non-incremental, quantifier-free constraints over bit-vectors and floating-point numbers, as in SLOT. These theories are central to program analysis and verification. For example, bit-vectors model low-level integer semantics in compiler intermediate representations (e.g., LLVM IR), machine code, and hardware circuits. Floating-point constraints capture the semantics of real-number approximations in numerical programs. Both are supported by the SMT-LIB standard and are widely implemented in modern SMT solvers. In § 7.1, we will discuss several directions for expanding the applicability to more SMT theories.

4 EXPERIMENTAL SETUP

This section describes the experimental setup used to evaluate the compiler optimizations-based SMT simplifications. To address these problems, we present TUNA(Figure 2), an evaluation framework for studying the interplay between compiler optimizations and SMT simplifications. We structure our evaluation methodology and results into two parts in the rest of this paper:

- TUNA-Opt: First, we collect a set of SMT queries, explore various compiler optimizations via search-based techniques (such as Genetic algorithm), and observe their impact on solver performance (**RQ1**).
- TUNA-Learn: We then train machine learning models to predict the optimal optimizations for unseen SMT queries based on features extracted from the input (**RQ2**).

Unless otherwise specified, we measure solver performance primarily by wall-clock solving time under a fixed timeout, and we additionally track satisfiability outcomes to ensure that the simplifications preserve semantics.
Compiler Optimizations. Our evaluation focuses on compiler optimizations applied to the LLVM IR translated from SMT formulas. We begin with 211 optimization passes documented by LLVM, and systematically filter out those unsuitable for our context. Specifically, we use the following criteria for excluding the passes:

- Utility passes (e.g., `dot-cfg`): they do not transform the LLVM IR.
- Module-level and loop passes: we build on the SLOT framework. which generates single-function, loop-free IR.
- Architecture-specific passes: they target low-level hardware features irrelevant to SMT solving.

Among the remaining candidates, many serve highly specialized purposes (e.g., targeting obfuscation, debugging, or profile-guided optimization) After this pruning, we identify 23 semantically meaningful and empirically impactful passes. These include `adce`, `dce`, and `instcombine`, which perform essential transformations such as eliminating dead code, simplifying instruction sequences, and propagating constants.

Evaluated SMT Solvers. We evaluate the impact of compiler optimizations on the performance of two mature, widely used, and state-of-the-art SMT solvers: Z3 and CVC5. They have broad support across various SMT theories, are widely used in academia and industry, and are actively maintained by developers. For consistency, we invoke each solver via a uniform command-line interface with a fixed timeout, and we record both runtime and returned status.

SMT-LIB2 Benchmarks. We analyze 83,963 constraints drawn from the non-incremental track of SMT-COMP 2024, following SLOT [57]. These constraints span three SMT logic categories: QF_BV (Quantifier-Free Bit-Vector), QF_FP (Quantifier-Free Floating Point), and QF_BVFP (Quantifier-Free Bit-Vector with Floating Point). They originate from various domains, including software validation, hardware verification, and cryptographic analysis. Constraint sizes range from 58 bytes to approximately 1.9 GB, and solver runtimes range from under 1 second to over 600 seconds. Several benchmark instances exhibit base solving times well beyond 600 seconds; however, this threshold is adopted as a practical timeout, consistent with SLOT [57].

Environment. All experiments are conducted on two servers, each equipped with two 56-core Xeon(R) Platinum 8176 CPU@2.10GHz and 500 GB of RAM, running Ubuntu 22.04. We pin each solver invocation to a single core to avoid cross-run interference and enable fair comparisons across optimization settings, while allowing multiple independent runs to execute in parallel across available cores.

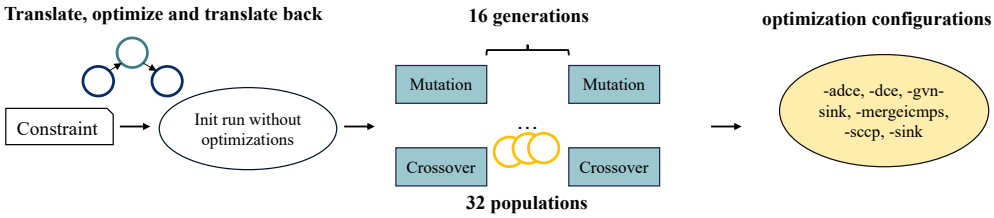


Fig. 3. Example of searching using a genetic algorithm

Table 3. Total CPU time (in minutes) consumed by the genetic algorithm (GA) for benchmarks in different solver runtime categories. Each category groups benchmarks by the time required by Z3 or CVC5 to solve them (e.g., "<1s" indicates benchmarks solved by the respective solver in under 1 second).

Benchmark	**<1s**		**1–30s**		**30–60s**		**60–120s**		**120–300s**		**300–600s**	
	Z3	CVC5	Z3	CVC5	Z3	CVC5	Z3	CVC5	Z3	CVC5	Z3	CVC5
QF_BV	2371	1906	3155	2983	2469	1005	2492	4533	5578	2827	2411	1463
QF_FP	895	666	52	178	85	156	272	143	488	257	376	224
QF_BVFP	432	427	204	245	200	41	79	103	116	21	177	54

5 COMBINED EFFECTS OF OPTIMIZATION FLAGS

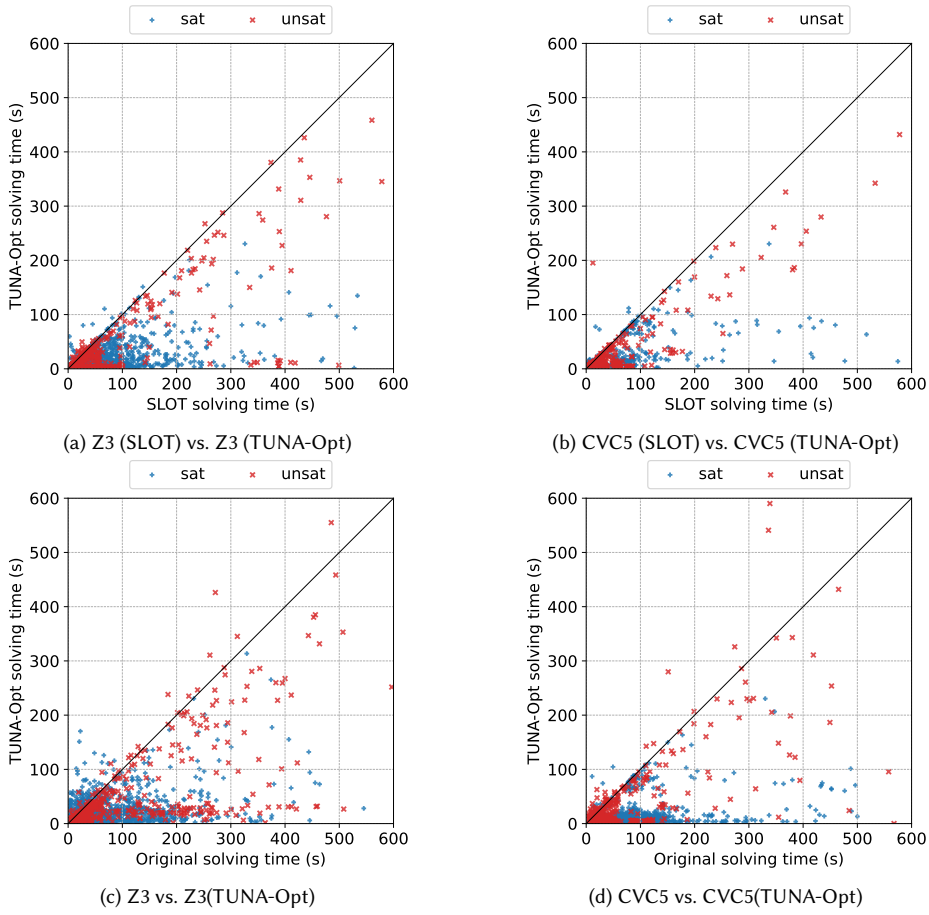
This section explores the combined effects of various compiler optimization flags on SMT solver performance. We aim to study whether tailored optimization strategies can significantly enhance solver efficiency across various SMT benchmarks. Given the many possible optimization combinations, determining effective ones for each SMT instance poses a significant challenge.

5.1 Methodology

Search-based Optimization. Given the vast and complex nature of the optimization space, characterized by high dimensionality and nonlinear interactions across optimization passes, an effective search strategy is essential for navigating this challenging landscape. After evaluating several approaches, we select a genetic algorithm (GA) for TUNA-Opt. GA is particularly well-suited for discrete, combinatorial search spaces due to its ability to explore diverse solutions and its robustness in handling non-linearities inherent in such domains.

As shown in Figure 3, TUNA-Opt maintains a fixed-size population of eight candidate solutions, each encoding a sequence of compiler optimization passes. In each generation, the top 25% of individuals, ranked by their impact on the SMT solver's performance, are selected to guide the generation of new candidates. When compiled with a given optimization sequence, fitness is defined as the solver's performance. The evolutionary process proceeds for a fixed number of generations. progressively identifying more effective optimization configurations.

Table 3 presents the CPU time consumed by the genetic algorithm (GA) for handling the SMT-LIB2 benchmarks in the QF_BV, QF_FP, and QF_BVFP categories. Benchmarks are grouped by the time required for Z3 or CVC5 to solve them, with categories ranging from under 1 second to 300–600 seconds. The reported CPU time corresponds to the cumulative time across all threads. Since GA executes in parallel, the actual wall-clock time is substantially lower, typically between 1% and 30% of the total CPU time.



(a) Z3 (SLOT) vs. Z3 (TUNA-Opt)

(b) CVC5 (SLOT) vs. CVC5 (TUNA-Opt)

(c) Z3 vs. Z3(TUNA-Opt)

(d) CVC5 vs. CVC5(TUNA-Opt)

Fig. 4. Scatter plot of per-query solving times under the baseline (x-axis) and TUNA-Opt (y-axis) on the full dataset of QF_BV, QF_FP, and QF_BVFP. Blue and red points denote unsatisfiable and satisfiable queries, respectively; points on the diagonal indicate equal performance.

Compared Baselines. We evaluate the impact of customized optimization pass selections by comparing them with several alternatives:

- **Default**: SMT solvers running without compiler-based simplifications serve as a control to measure the raw solving performance.
- **O3**: The default optimization level in LLVM applies a general-purpose set of optimizations to improve overall program performance, but is not tailored for SMT solvers.
- **SLOT** [57]: The state-of-the-art work that applies several fixed optimization passes for SMT formulas based on empirical observations.

5.2 Results and Analysis

This section evaluates the impact of combined compiler optimizations on SMT solver performance. We first present the results of the best-identified optimization configurations from the genetic search (§ 5.2.1). We then analyze the influence of optimization settings in two ways: (1) by comparing the best and worst configurations for individual constraints, highlighting the intra-formula performance variance (§ 5.2.2) and (2) by assessing the structural similarities of effective configurations across constraints (§ 5.2.3).

5.2.1 Best Optimization Configuration Performance. We begin by evaluating the performance of the best optimization configurations generated by TUNA-Opt in comparison to the baselines. For clarity, we overload the term TUNA-Opt to refer to the variant instantiated with these best-performing configurations.

Overall Performance Comparison. Figure 4 presents a comparative analysis of TUNA-Opt against two baselines—SLOT and the default configuration—using the Z3 and CVC5 SMT solvers.

TUNA-Opt outperforms both baselines across a majority of the queries. It reduces solving time to under 100 seconds for most queries, whereas SLOT frequently exceeds 200 seconds. Compared to the default configuration, TUNA-Opt yields substantial speedups, often reducing solve times from over 200 to under 100 seconds, with improvements ranging from 2× to 5×. The concentration of points below the diagonal and in the lower-right quadrant of the scatter plots highlights TUNA-Opt's advantage, especially on complicated queries. These results show the effectiveness of the genetic search for optimizing configurations.

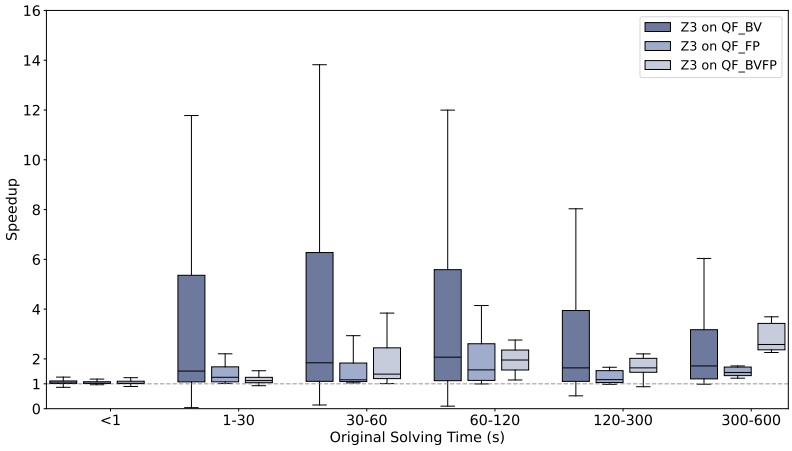
Comparison with SLOT. Figure 5 compares TUNA-Opt and SLOT across a range of constraint complexities and various benchmark sets. On Z3, TUNA-Opt achieves substantial speedups—exceeding 2× in the median in QF_BV benchmarks. Similar trends hold for QF_FP and QF_BVFP, though with slightly reduced variance. On CVC5, TUNA-Opt continues to outperform SLOT, particularly in the 60–300s baseline solving time range, where it consistently delivers over 1.5× speedups with lower variance. These results show the effectiveness of the adaptive optimization strategy TUNA-Opt, driven by a genetic algorithm, in improving solver performance in diverse back-ends and logics.

For short queries (solved in less than one second), both TUNA-Opt and SLOT offer only marginal performance improvements. This is primarily because such queries inherently present limited opportunities for optimization: (1) The solver already performs near its peak efficiency, leaving little room for further acceleration; (2) The fixed overheads associated with the optimization pipeline—such as converting the input into LLVM IR, applying transformations, and re-emitting SMT-LIB2 code—constitute a significant portion of the total execution time in these low-latency cases. As a result, even when optimizations are applied, their benefits are mainly masked by these fixed costs.

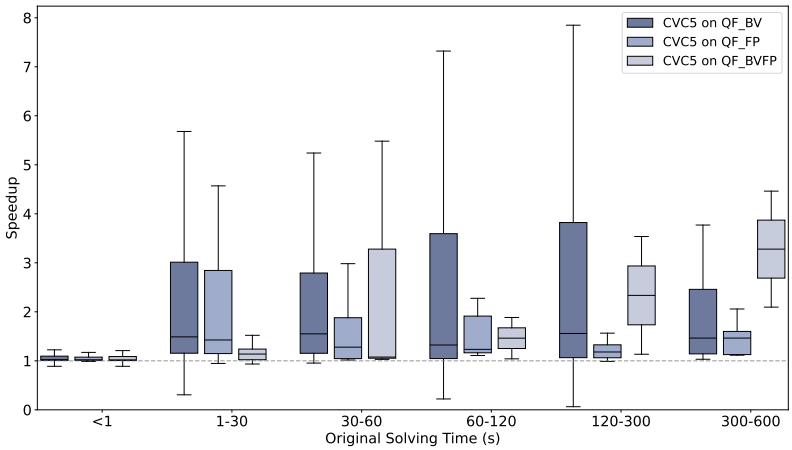
Comparison with O3. To assess the efficacy of domain-specific optimization relative to general-purpose compiler strategies, we compare TUNA-Opt against LLVM's -O3 optimization level.

As shown in Figure 6, both TUNA-Opt outperform O3 across a wide variety of instances. On Z3, TUNA-Opt yields speedups predominantly in the range of 1.2× to 1.8×, with a median of approximately 1.45× and a maximum near 2.4×. SLOT performs slightly worse, with a median of around 1.3×. On CVC5, TUNA-Opt achieves a median speedup of approximately 1.7×, with some benchmarks exceeding 2.7×; SLOT again trails slightly, with a median just under 1.5×. While O3 occasionally matches domain-specific techniques on trivial or degenerate inputs, it can underperform on more complex SMT queries.

This disparity arises from O3's design, which targets native execution performance through general-purpose optimizations such as loop unrolling and aggressive inlining. While effective for traditional compilation, these transformations



(a) Z3



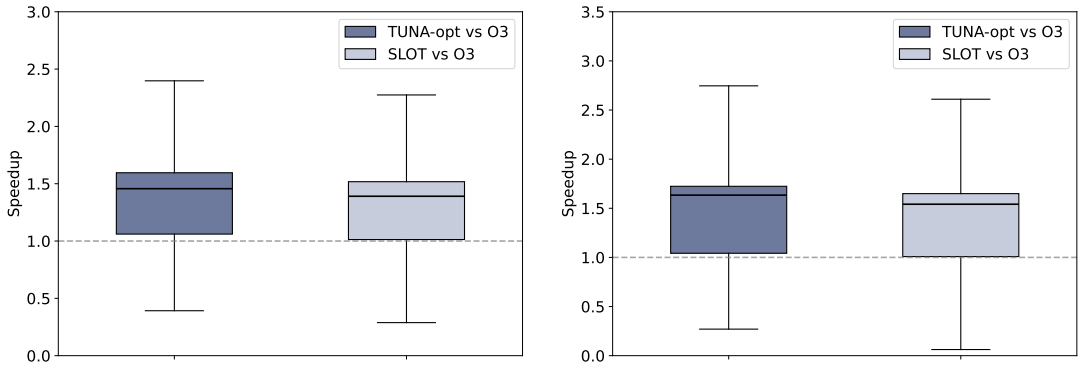
(b) CVC5

Fig. 5. Speedup distribution of TUNA-Opt relative to SLOT for each benchmark set under Z3 (5a) and CVC5 (5b), grouped by the original solving time of the constraint (x-axis).

can be detrimental in SMT solving, where increased code size and structural complexity hinder symbolic reasoning. In contrast, TUNA-Opt applies domain-aware transformations, yielding more robust performance gains across solvers and benchmarks.

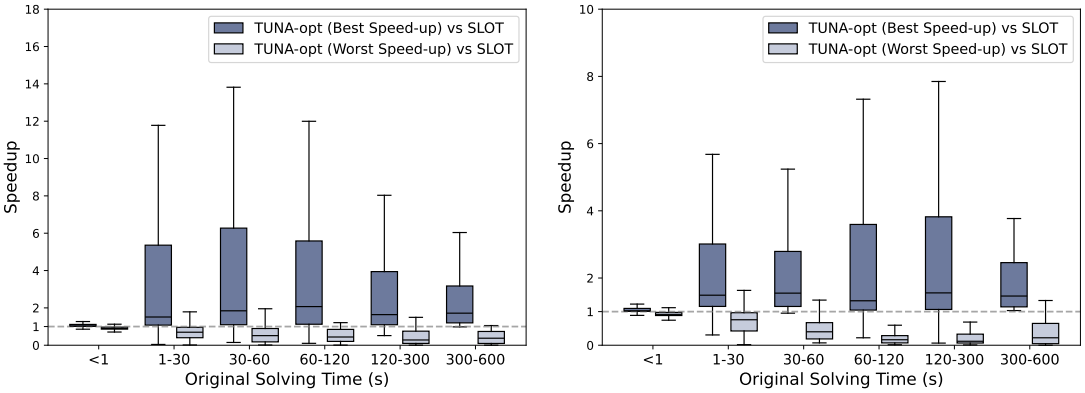
5.2.2 Intra-Constraint Performance Variance. This subsection shows that compiler optimization sequences can cause substantial performance variation for a fixed SMT constraint. Specifically, we compare the best and worst optimization configurations identified by the generic algorithm in TUNA-Opt to highlight this variability.

Figure 7 illustrates the distribution of speedups achieved by the best and worst optimization configurations produced by TUNA-Opt, relative to SLOT, across different ranges of solver times. The evaluation reveals that Z3 and CVC5 exhibit significant intra-instance performance variability across compiler optimization configurations. For Z3, across



(a) Z3 (b) CVC5

Fig. 6. Speedup distribution of TUNA-Opt and SLOT relative to O3. Each box represents the distribution of speedups across the full dataset, including benchmarks from QF_BV, QF_FP, and QF_BVFP.

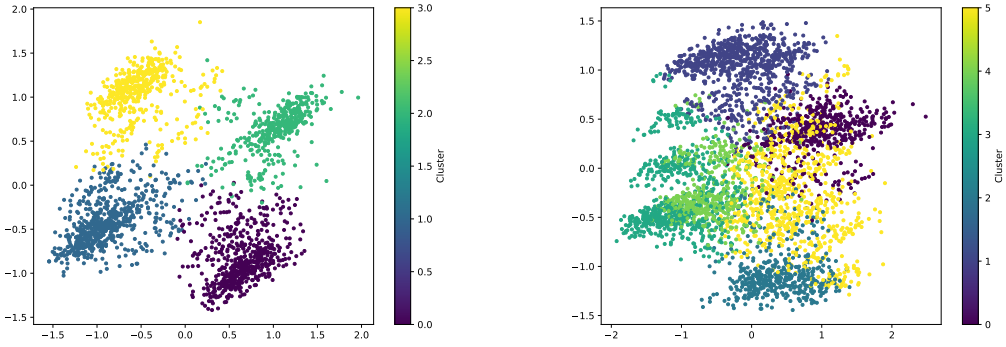


(a) Z3 (b) CVC5

Fig. 7. Speedup distribution of best TUNA-Opt configuration and worst TUNA-Opt configuration relative to SLOT. Each box represents the distribution of speedups for benchmarks from the QF_BV, QF_FP, and QF_BVFP grouped by their original solving time.

the medium to high complexity range (30–300s), the best configurations achieve 2–3× speedups over SLOT, with some cases exceeding 6×. In contrast, the worst configurations can reduce performance to below 1×, sometimes halving it. CVC5 shows similar trends, with the largest divergence in the 60–120s range, where the best configuration yields a 7× speedup, and the worst degrades performance to 0.4×. In the low-complexity range (<1s), the performance gap between the best and worst configurations narrows significantly. This is primarily due to the dominance of fixed system overheads, such as input parsing and IR construction, relative to the short solving time, limiting the observable impact of compiler optimizations.

These findings reinforce that the optimization sequence critically influences solver performance for a fixed SMT query. While effective configurations can markedly reduce solving time, poorly chosen sequences may incur overhead



(a) k-means cluster of Z3

(b) k-means cluster of CVC5

Fig. 8. Clustered distributions of optimal LLVM pass configurations for Z3 and CVC5. Each point represents the configuration selected for a specific SMT instance, projected onto two dimensions using PCA.

and degrade performance. Besides, fixed strategies such as SLOT or O3 can miss opportunities to capture and utilize instance-specific variability.

5.2.3 Cross-Constraint Optimization Configuration Similarity. This subsection analyzes the structural commonalities in optimal configurations found by TUNA-Opt across SMT instances.

Clustering Analysis of Optimal Pass Configurations. We analyze structural similarities among per-instance optimal configurations by clustering them in a 23-dimensional space, where each dimension corresponds to an LLVM optimization pass. To identify recurring configuration patterns, we apply the k-means clustering algorithm. The optimal number of clusters—determined via the elbow method applied to the within-cluster sum of squares—is 4 for Z3 and 6 for CVC5. To visualize the clusters, we project the clustered configurations into two dimensions using Principal Component Analysis (PCA). Figure 8 visualizes the resulting clusters, with each point representing the optimal configuration for a single SMT instance. The emergence of distinct clusters indicates that structurally similar optimization strategies recur across semantically diverse SMT queries, revealing common patterns in effective compiler configurations.

Pass Usage Analysis Across Clusters. Figure 9 presents a heatmap illustrating the frequency of LLVM passes across clusters identified for Z3 and CVC5. The horizontal axis enumerates individual LLVM passes; the vertical axis corresponds to cluster indices. Darker cells indicate higher pass frequencies within a cluster. The heatmap reveals clear distinctions in optimization strategies across clusters. For example, CVC5 Cluster 1 frequently selects passes such as `gvn`, `mergeicmps`, and `instcombine`, whereas Cluster 2 favors lighter transformations such as `reassociate`. These differences reflect heterogeneous optimization preferences among SMT instances. Nonetheless, several passes—most notably `instcombine`, `gvn`, and `reassociate`—appear consistently across clusters, suggesting their broad utility.

Representative Configurations Within Clusters. Tables 4a and 4b enumerate each cluster's most representative pass configurations. These configurations, derived from frequency analysis, characterize the dominant optimization strategies within each group. For Z3, Cluster 2 combines structural simplifications (`bdce`, `dse`) with canonicalization (`instcombine`, `reassociate`) and scalar optimizations (`sroa`, `sccp`). In contrast, CVC5 clusters exhibit more diverse

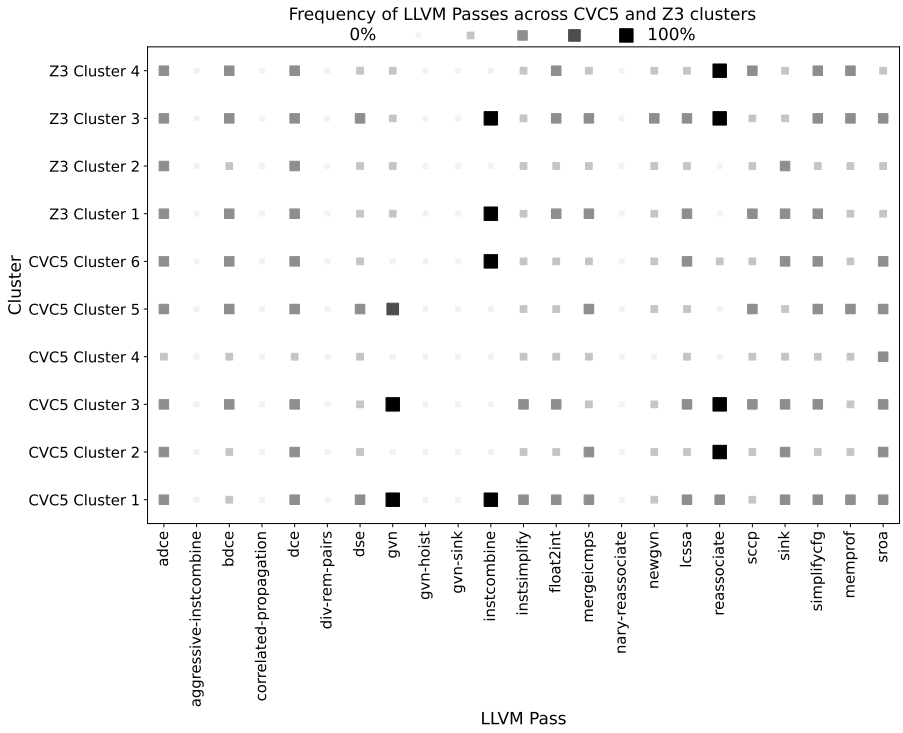


Fig. 9. LLVM pass application frequency across identified configuration clusters for Z3 and CVC5. Darker colors indicate a higher frequency of pass usage within each cluster.

configurations, ranging from aggressive transformations (e.g., Cluster 0 with `gvn`, `mergeicmps`) to minimal sequences (e.g., Cluster 3 with only `sink`).

These results indicate that effective optimization sequences are neither uniform nor randomly distributed. Instead, they exhibit structural regularities aligned with solver-specific and instance-specific characteristics. This observation motivates the use of predictive models to infer suitable optimization configurations based on instance features.

5.3 Summary

The results demonstrate that SMT solving performance is highly sensitive to the choice of LLVM optimization configuration, with effectiveness varying substantially across instance characteristics and transformation strategies. First, TUNA-Opt outperforms both the default LLVM configuration and the SLOT baseline, yielding 3–5× average speedups and exceeding 10× in select cases, particularly for medium- and high-complexity constraints. Second, the standard `-O3` optimization level—designed for general-purpose code generation—often impairs solver performance by applying transformations that obscure logical structure. Third, the impact of optimization is nontrivial: for a fixed instance, the choice of configuration can induce order-of-magnitude differences in solving time. Finally, clustering analysis reveals structural regularities among effective configurations, indicating the presence of generalizable patterns. These findings motivate the use of learning-based methods to automate optimization selection based on instance features.

Table 4. Frequently occurring LLVM pass configurations within each cluster for Z3 and CVC5. Each row lists the most representative passes characterizing a configuration cluster.

(a) Z3

Cluster	Passes
1	adce, dse, instcombine, memprof, sccp
2	adce, dce, float2int, sccp
3	bdce, dse, float2int, instcombine, lcssa, reassociate, sccp, sink, sroa
4	memprof, newgvn, reassociate, sccp

(b) CVC5

Cluster	Passes
1	bdce, gvn, instcombine, mergeicmps, reassociate, simplifycfg
2	float2int, gvn, reassociate
3	bdce, dce, dse, gvn, instcombine
4	sink
5	adce, mergeicmps, sccp, simplifycfg, sroa
6	float2int, gvn, lcssa, memprof

Answer to RQ1: SMT solver performance is highly sensitive to compiler optimizations. Carefully selected configurations yield up to 10× speedups, while general-purpose levels such as `-O3` may degrade performance. The most effective configurations exploit synergistic transformations aligned with instance structure.

6 MACHINE LEARNING-BASED OPTIMIZATION PREDICTOR

In this section, we explore machine learning-based predictors for automatically selecting optimizations based on the characteristics of the SMT instance. Once trained, these models can predict which optimizations will yield the best performance without requiring the computationally expensive iterative searches.

6.1 Methodology

As illustrated in Figure 2, we collect execution logs from RQ1 for various SMT instances, capturing key performance metrics and applying optimization passes. We derive class labels from each SMT instance's optimized optimization configurations. We train machine-learning classifiers to predict effective optimization passes using these labels and features extracted from the problem instances.

6.1.1 Problem Features. Selecting the right features for the predictive model is crucial for its effectiveness. We design the features based on two primary criteria: computational efficiency and relevance to the prediction task. First, feature extraction should be efficient enough to avoid excessive overhead. Second, the selected features must capture essential characteristics of the problem instance to ensure meaningful predictions.

IR2Vec Embedding. The first approach we employ for feature extraction is the IR2Vec embedding [81]. This method converts the Intermediate Representation (IR) of code into continuous vectors, which serve as input to machine learning models. The intuition behind IR2Vec is that programs with similar IR structures will produce embeddings that are close to each other in the vector space. Specifically, the IR2Vec embedding uses two encoding strategies: symbolic and flow-aware. Symbolic encoding focuses on seed embedding, whereas flow-aware encoding augments it with control-flow

information. To standardize these embeddings, we apply two normalization steps: scaling each vector element to the range 0 to 1 and normalizing each coordinate across the dataset.

Handcrafted IR Features. In addition to IR2Vec embeddings, we manually design features to capture key metrics of LLVM IRs translated from SMT problems. We implement an extractor to extract features during the SMT-to-IR transformation, traversing the translated SMT function module. These include (1) *Instruction metrics* such as counts of arithmetic, logical, and memory operations, etc.; (2) *Structural features* such as characteristics of basic blocks (e.g., number of predecessors and successors), instruction density, phi-node configurations, etc. Table 8 presents the complete list of handcrafted features. These features complement the IR2Vec embeddings, providing high-level and structural representations.

SMT-LIB2 Features. During the translation from SMT-LIB2 inputs to LLVM IRs, some important structural information may be lost, hampering the learner's effectiveness. Hence, beyond LLVM IR-level features, we also incorporate SMT-LIB2-level features to capture the characteristics of SMT constraints. Specifically, we extract abstract syntax tree (AST) features from SMT formulas using the pysmt library, following the prior work [53]. These include the number of assertions, the depth of the AST tree, the number of nodes of the AST tree, the number of different types of operands, etc. As the extraction of SMT-LIB2 features in [53] only supports the QF_BV benchmark, we only show its performance on QF_BV in the following.

6.1.2 Predicting the Optimizations. We aim to predict effective optimization passes for a given input formula to enhance the solver's performance. This task is formulated as a *multi-label classification problem*. Specifically, for an input formula I_i, the predictor produces a binary vector $\hat{\mathbf{y}}_i = (\hat{y}_{i1}, \hat{y}_{i2}, \ldots, \hat{y}_{iN})$, where each $\hat{y}_{ij} \in \{0, 1\}$ indicates whether the j-th optimization pass is recommended for that formula. The selected passes collectively influence the solver's efficiency.

The Training Set. The training dataset is constructed using a subset of SMT formulas $\{I_1, I_2, \ldots, I_M\}$, derived from the experiments in RQ1. For each formula I_i, a *Genetic Algorithm (GA)* is employed to explore the configuration space of optimization passes. The optimal configuration identified for each formula is represented as a binary vector $\mathbf{y}_i = (y_{i1}, y_{i2}, \ldots, y_{iN})$, where $y_{ij} = 1$ if the j-th optimization pass belongs to the optimal set, and $y_{ij} = 0$ otherwise. This binary vector serves as the ground truth label for training the predictor.

Threshold-Based Instance Labeling. Rather than relying solely on the single best-performing configuration, we select a set of high-performing configurations. Specifically, configurations are included if they achieve at least 90% of the maximum observed performance improvement. This approach has two primary benefits. First, it reduces the risk of overfitting to noisy or outlier configurations by considering a broader range of effective configurations. Second, it mitigates the impact of measurement noise and variability inherent in the optimization process, thereby enhancing the robustness of the training labels.

Delta Debugging-inspired Label Pruning. We adopt a delta debugging-inspired approach for each instance to isolate the optimization passes responsible for a given performance behavior. Starting from the full configuration labels empirically associated with the observed speedup, our approach iteratively removes passes. To mitigate the confounding effects of execution-time variability, we use the IR structure as the reduction oracle. Specifically, a pass is considered removable only if its elimination does not alter the resulting IR. After this step, we could obtain a reduced subset of passes that still produce the same performance speedups.

The Classifiers. Next, we present how to train the classifiers to predict effective optimizations. Our approach is grounded in the simplifying assumption that each optimization pass can be predicted independently. While this

assumption does not hold universally—certain passes may exhibit complex interactions—it provides a useful baseline for understanding how well a purely independent approach predicts optimization effectiveness before extending the method to account for potential dependencies among passes.

We evaluate a range of standard classification models, including decision trees (DT), random forests (RF), k-nearest neighbors (kNN), ridge classifiers (RC), support vector machines (SVM) via the SMO algorithm, and multilayer perceptrons (MLP). This workflow reflects a common experimental setup used in prior empirical studies (e.g., [85], [94]). Our goal is to assess the adaptability of TUNA-Learn across diverse model architectures and to investigate its effectiveness in predictive tasks. For each model, we apply grid search to tune hyperparameters.

Cross-Validation. We employ 3-fold cross-validation to ensure robust generalization of the model. This method divides the dataset into three folds, each used as a test set once, while the remaining two are used for training. This process is repeated three times to ensure that each fold acts as the test set exactly once.

6.1.3 Metrics. In alignment with LEO [18], SLOT [57], and prior work on compiler auto-tuning, we use the end-to-end performance to evaluate the performance of TUNA-Learn. Specifically, we follow SLOT [57], using geometric mean speedup to evaluate the performance. In addition, we use the metric Relative Performance Match (RPM) to measure ToolName-Learn's ability to match the optimal performance of TUNA-Opt.

Relative Performance Match. To evaluate how closely TUNA-Learn approximates the optimal configurations identified by TUNA-Opt, we introduce the *Relative Performance Match* (RPM) metric. RPM quantifies the proportion of instances where the predicted configuration achieves performance within a specified factor of the optimal.

Formally, let t_i^{opt} and t_i^{learn} denote the execution time of instance i under the optimal configuration (as determined in RQ1) and the configuration predicted by TUNA-Learn (RQ2), respectively. For a threshold $\tau \in (0, 1]$, we define:

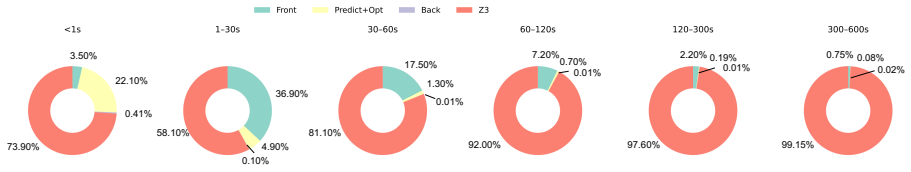
$$\text{RPM}_{\text{time}}(\tau) = \frac{1}{N} \sum_{i=1}^{N} \mathbb{1} \left[\frac{t_i^{\text{learn}}}{t_i^{\text{opt}}} \leq \frac{1}{\tau} \right]$$

This metric captures the fraction of instances for which the predicted configuration yields execution time within a $1/\tau$ factor of the optimal execution time. Higher values of τ correspond to stricter performance requirements.

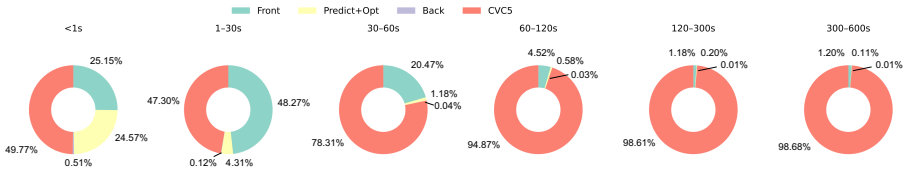
6.2 Results and Analysis

This section presents a detailed evaluation of our machine learning-based predictor, organized into four primary components. We begin by analyzing the computational overhead introduced by incorporating machine learning into the solver pipeline, examining both the offline costs of model training and the online costs of prediction (§ 6.2.1). Following this, we compare TUNA-Learn with TUNA-Opt to understand the performance gap between learned and iteratively searched optimization configurations (§ 6.2.2). Next, we assess how TUNA-Learn performs relative to default solver configurations and the SLOT framework, considering both standalone and portfolio-based methods(§ 6.2.3). These comparisons use the decision tree model with the IR2Vec features, denoted TUNA-Learn (DT + IR2Vec), which offers the best trade-off between training cost and predictive precision (see Table 7). Finally, we conduct an ablation study to isolate the contributions of various machine learning models and features (§ 6.2.4).

6.2.1 TUNA-Learn Overhead. Before delving into performance comparisons, we evaluate the computational overhead incurred by integrating machine learning into the solver workflow. These overheads arise during both the offline model training phase and the online prediction phase.



(a) Z3



(b) CVC5

Fig. 10. Overhead of for TUNA-Learn across different time ranges on full dataset of QF_BV, QF_FP and QF_BVFP

Table 5. Training time (in seconds) of different model-feature combinations for Z3 and CVC5.

Features	**DT**		**kNN**		**RC**		**MLP**		**RF**	
	Z3	CVC5	Z3	CVC5	Z3	CVC5	Z3	CVC5	Z3	CVC5
Handcrafted	46.64	56.37	21.48	18.46	3.61	6.67	3256.57	4392.67	28.84	147.67
IR2Vec	514.26	1034.81	10.57	5.93	10.28	10.28	1958.67	3276.51	330.00	280.45
SMT	25.98	29.88	11.49	16.48	3.28	5.80	2931.80	3427.01	55.02	111.20

Table 5 reports offline training times across different model-feature configurations. Training costs vary substantially by model class. Multi-layer perceptron (MLP) incurs the highest cost, with training times exceeding 3000 seconds in some settings. In contrast, decision trees (DT), random forests (RF), and ridge classifiers (RC) train considerably faster, typically completing in 500 seconds or less.

To assess prediction-time overhead, we measure the additional cost introduced by formula translation, feature extraction, and model prediction across a representative benchmark suite. As shown in Figure 10, this overhead is negligible for non-trivial queries, where solver runtimes dominate. In these cases, the performance gains from learned guidance outweigh the cost of prediction. However, for trivial queries—those solvable in under one second—the overhead becomes significant, contributing 22.1% of total runtime in Z3 and 26.2% in CVC5. We discuss these trade-offs in more detail in § 7.3, where we identify scenarios in which prediction overhead may offset its benefits.

6.2.2 Comparison with TUNA-Opt. This section compares the performance of TUNA-Learn and TUNA-Opt using the RPM metric. Higher RPM values at larger τ thresholds reflect closer alignment with the performance of TUNA-Opt. We report results using the IR2Vec feature representation with the DT model, which consistently outperforms other configurations (Table 7) and demonstrates its effectiveness for learning-based tasks.

Table 6 illustrates the distribution of optimization effectiveness across different thresholds of τ—the ratio of TUNA-Learn's solving time to that of TUNA-Opt. The key observations are as follows:

Table 6. Distribution of optimization effectiveness for TUNA-Learn (DT+IR2Vec), where each bar segment represents RPM, the percentage of cases achieving specific τ (100%, 80%, 60%, 40%, 20%) compared to the TUNA-Opt solving time. A larger proportion in higher percentage ranges indicates better optimization performance, showing how each model maintains TUNA-Opt's original solving capability while reducing computation time.

Solver\RPM	**20%**	**40%**	**60%**	**80%**	**100%**
Z3	0.02	0.04	0.17	0.36	0.41
CVC5	0.13	0.07	0.14	0.27	0.49

- For both solvers, the highest RPM proportion occurs at $\tau = 100\%$, with TUNA-Learn achieving 0.41 for Z3 and 0.39 for CVC5. This suggests that the learning-based model can closely replicate TUNA-Opt's performance in a significant number of cases.
- The fraction of cases where TUNA-Learn leads to performance degradation (i.e., $\tau \leq 40\%$) is notably low—only 6% for Z3 (0.04 at 40% and 0.02 at 20%) and 20% for CVC5 (0.07 at 40% and 0.13 at 20%). This reflects a relatively low risk of harmful predictions, especially for Z3.
- TUNA-Learn displays more stable performance for Z3, with a higher concentration in the top τ ranges and fewer detrimental cases, highlighting its effectiveness in capturing optimization-relevant features for that solver.

These findings highlight a correlation between optimization configurations and instance-specific features. While the learning-based approach does not consistently match the performance of iterative search—an expected limitation given the models' difficulty in capturing complex interactions among passes—it nonetheless provides practical benefits, as we will demonstrate below.

6.2.3 Comparison with Default Solvers and SLOT. In this section, we present the performance of our best-performing model TUNA-Learn (DT+IR2Vec).

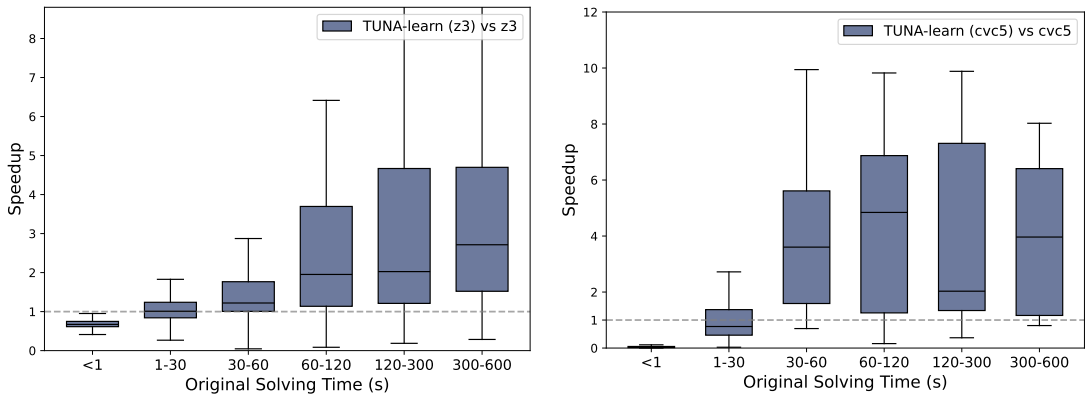
In practical deployment scenarios, we assess the portfolio performance of TUNA-Learn by executing it concurrently with the default solver. In this configuration, both the TUNA-Learn pipeline and the default solver are run in parallel, and the result returned first, whether by TUNA-Learn or by the default solver, is adopted. We define T_{default} as the time the default solver requires to conclude either sat or unsat for a given benchmark. The total runtime of TUNA-Learn consists of four components: extraction of features and prediction time T_{pred}, optimization time T_{opt}, translation time T_{trans}, and solving time with the selected backend T_{solve}. In alignment with SLOT [57], the portfolio runtime is defined as $\min(T_{\text{default}}, T_{\text{TUNA-Learn}})$, where $T_{\text{TUNA-Learn}} = T_{\text{pred}} + T_{\text{opt}} + T_{\text{trans}} + T_{\text{solve}}$.

Following the SLOT evaluation methodology, we first assess the standalone performance of TUNA-Learn, the default solvers, and SLOT independently. We then measure performance under the portfolio method.

Standalone Performance. We first compare the standalone performance of TUNA-Learn (DT+IR2Vec) with default solvers and SLOT.

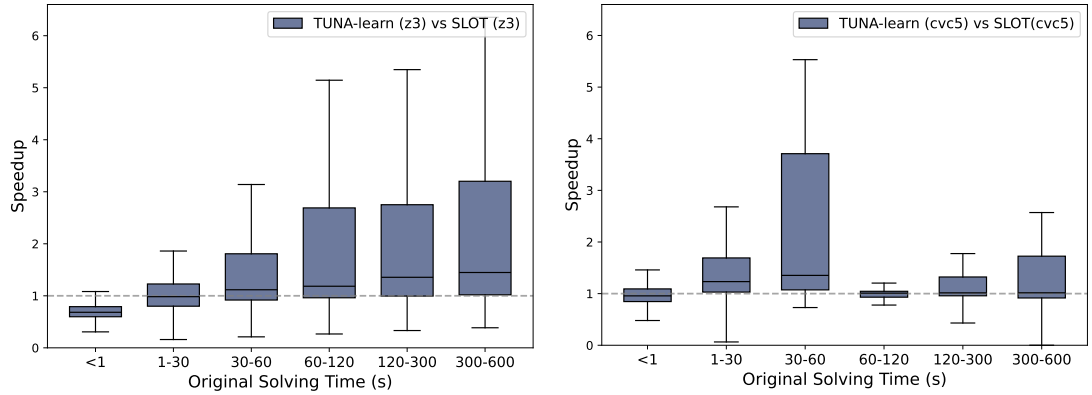
Standalone vs. Default. In Figure 11, the performance improvements without the portfolio method of TUNA-Learn (DT+IR2Vec) over default solver settings are reported. Two trends stand out.

- First, while Z3 exhibits increasing speedups with constraints longer than 30s, CVC5 shows more steady improvements with constraints longer than 30s. This discrepancy may stem from solver-specific pass sensitivity or differences in the effectiveness of optimization passes exposed by the learning model. For example, TUNA-Learn (DT+IR2Vec) achieves median speedups exceeding 30s for constraints, with a growing trend, and the highest



(a) Z3 (b) CVC5

Fig. 11. Comparing default solvers and TUNA-Learn (DT+IR2Vec) with data from QF_BV, QF_FP and QF_BVFP.



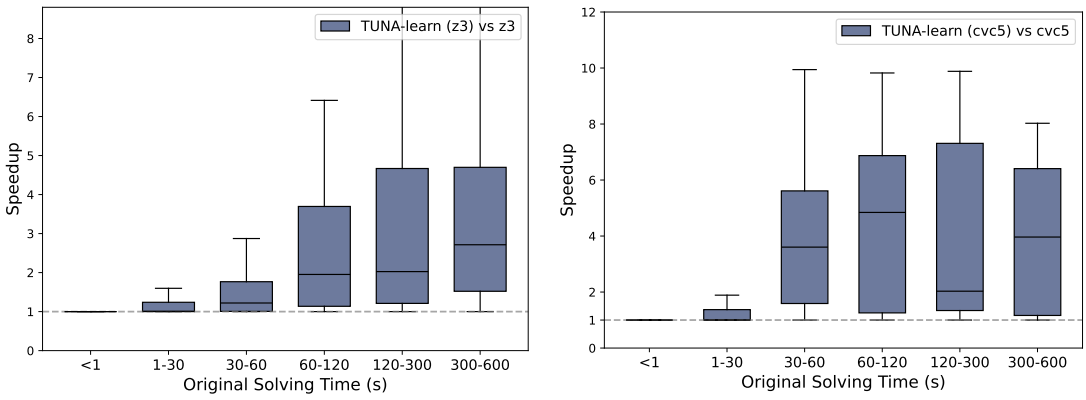
(a) Z3 (b) CVC5

Fig. 12. Comparing SLOT and TUNA-Learn (DT+IR2Vec) for speeding up Z3 and CVC5, with data from QF_BV, QF_FP and QF_BVFP.

median speedup is nearly 3×. While CVC5 shows speedups generally above 2s for constraints exceeding 30s, it decreases for constraints in the time interval 120s-300s.

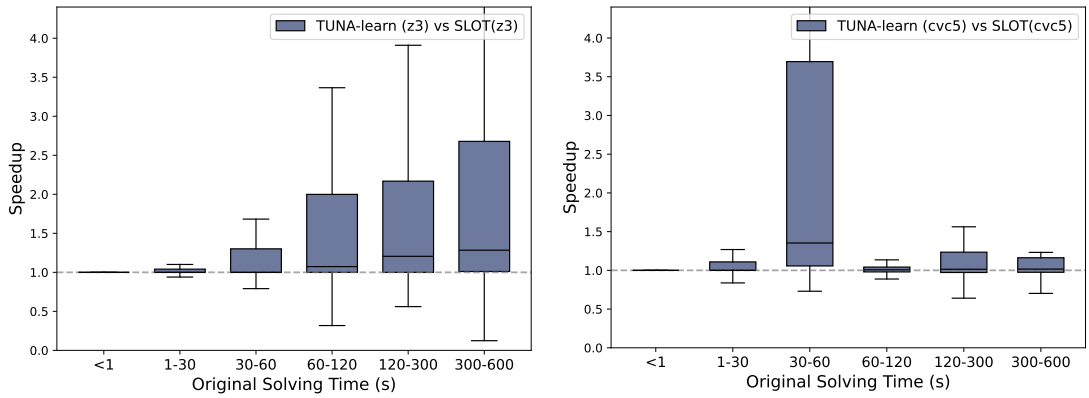
- Second, the gains are most pronounced on constraints with baseline solve times exceeding 30 seconds. Like TUNA-Opt, the learned configuration is more effective on complex inputs, increasing speedups as baseline difficulty grows, with median speedup significantly above 1.

Standalone vs. SLOT. Figure 12 compares the speed-ups our predictor and SLOT achieve across various SMT instances. While SLOT remains a strong baseline, TUNA-Learn (DT+IR2Vec) performs relatively better. For Z3, TUNA-Learn (DT+IR2Vec) yields noticeable improvements on constraints with baseline solving times exceeding 1 second. Both the median speedup and the overall distribution of gains increase with constraint complexity. In contrast, for constraints solvable in under 1 second, the optimization potential is limited because translation and prediction overheads dominate,



(a) Z3 (b) CVC5

Fig. 13. Comparing default solvers and TUNA-Learn (DT+IR2Vec, Ptf) using portfolio method, with data from QF_BV, QF_FP, and QF_BVFP.



(a) Z3 (b) CVC5

Fig. 14. Comparing SLOT (Ptf) and TUNA-Learn (DT+IR2Vec, Ptf) for speeding up Z3 and CVC5, with data from QF_BV, QF_FP and QF_BVFP, using portfolio method.

diminishing returns. For CVC5, speedups generally remain above 1, with the highest median improvements observed in the 30–60 second range.

Portfolio Performance. We compare the portfolio performance of TUNA-Learn against both the default solvers and SLOT, and further contrast these results with their respective standalone performance.

Portfolio vs. Default. Figure 13 illustrates the performance of TUNA-Learn (DT+IR2Vec) under the portfolio setting, denoted as TUNA-Learn (DT+IR2Vec, Ptf), compared against the default solvers Z3 and CVC5.

First, this approach retains the strong performance of TUNA-Learn (DT+IR2Vec, Ptf) on more complex constraints, while also compensating for its deficiencies on simpler cases. The improvement becomes increasingly pronounced as constraint complexity grows. In particular, for Z3, the median speedup steadily rises beyond the 30s threshold, peaking

near 3× in the 300–600s range. Similarly, for CVC5, the speedup improves significantly for constraints in the 60–300s range, reaching above 5× at its best. The overall distribution of speedups improves for both Z3 and CVC5.

Second, even in the <30s range, where standalone TUNA-Learn struggled due to overhead, the portfolio setup ensures performance remains on par with, or better than, the baseline solver. This demonstrates that the portfolio method not only complements TUNA-Learn on simple cases but also amplifies its advantage on difficult benchmarks.

Overall, the strategy yields robust improvements across the full spectrum of problem difficulty. In particular, TUNA-Learn's performance for solving complex constraints is generally improved. This approach preserves ToolName-Learn's strong performance on harder constraints while effectively mitigating its limitations on trivial ones.

Portfolio vs. SLOT. In the portfolio setting (Figure 14), we compare SLOT (Ptf) and TUNA-Learn (DT+IR2Vec, Ptf), where both techniques are executed in parallel with default solvers. Two key observations emerge:

First, for trivial queries, both SLOT (Ptf) and TUNA-Learn (DT+IR2Vec, Ptf) yield speedups close to 1×, indicating that translation or prediction may introduce overhead to both approaches, slightly delaying solution time relative to the default solvers. However, for both Z3 and CVC5, TUNA-Learn (DT+IR2Vec, Ptf) consistently outperforms SLOT on instances with baseline solving times exceeding 1 second. The advantage is particularly pronounced for Z3 on harder queries. For example, in the 120–300s interval, TUNA-Learn (DT+IR2Vec, Ptf) achieves a median speedup of approximately 1.2× over SLOT.

Second, the performance gap increases with constraint complexity, especially for Z3. Although SLOT (Ptf) significantly improves performance on difficult queries, TUNA-Learn (DT+IR2Vec, Ptf) achieves even greater gains. In the 300–600s range, it reaches a median speedup of 1.3×, demonstrating the effectiveness of learned pass selection in guiding solver behavior. For CVC5, the trend is less marked; the highest median speedup of roughly 1.4× occurs in the 30–60s interval, indicating modest but consistent improvements. For queries exceeding 60 seconds, the additional benefit over SLOT (Ptf) is limited.

Overall, these results show that TUNA-Learn performs competitively with SLOT in the portfolio setting, effectively leveraging learned optimizations and solver diversity to improve performance across a range of query complexities.

6.2.4 Studying the Impact of Classifiers and Features. This section presents an ablation study evaluating the impact of different feature sets and classifiers on solver performance. We compare models trained with IR2Vec, handcrafted, SMT-LIB2, and a combination of IR2Vec and SMT-LIB2 features. The evaluation is conducted using the portfolio method over default solvers.

Table 7 shows the speedups of different models (DT, kNN, RC, MLP, and RF) and different features (Handcrafted, IR2Vec, SMT-LIB2, and IR2Vec+SMT-LIB2) in predicting the pass configuration. To improve clarity, we merge the previously separate 30s and 60s intervals into a single 60s threshold. The empirical results show minimal variation in the solver's and the model's behavior between 30 and 60 seconds, making a finer distinction unnecessary. This consolidation simplifies the analysis and better aligns time ranges with observed performance transitions.

Impact of ML Models. All the models show performance gains across time intervals. Specifically, the DT classifier emerges as the most effective model overall, achieving the highest RPM of 0.41 and 0.39 with $\tau = 100\%$ for the solvers Z3 and CVC5, respectively, when using IR2Vec features. Additionally, DT achieves notable average geometric mean speedups of 1.15× for Z3 and 1.54× for CVC5. This trend is consistently observed across different timeout intervals. The DT model not only achieves the highest geometric speedups in the long-timeout ranges (e.g., 3.28× for Z3 and 3.29× for CVC5 in the 300–600s range) but also maintains relatively strong and stable performance across all time brackets.

Table 7. Geometric mean speedup of models using different feature representations across time intervals in the portfolio setting, relative to default solvers. Evaluation is performed on test instances from QF_BV, QF_FP, and QF_BVFP.

Model	Features	Z3 Speedup					CVC5 Speedup				
		<1	1–60	60–120	120–300	300–600	<1	1–60	60–120	120–300	300–600
DT	Handcrafted(IR)	1.00	1.03	1.24	1.33	1.62	1.00	1.17	1.49	1.31	1.34
	IR2Vec(IR)	1.00	1.19	2.23	2.57	**3.28**	1.00	1.39	3.29	2.74	**2.83**
	SMT-LIB2	1.00	1.04	1.31	1.44	1.73	1.00	1.06	1.44	1.34	1.50
	Combined	1.00	1.09	1.27	1.50	1.69	1.00	1.05	1.36	1.67	2.29
kNN	Handcrafted(IR)	1.00	1.02	1.25	1.39	1.33	1.00	1.10	1.40	1.25	1.33
	IR2Vec(IR)	1.00	1.03	1.35	1.40	1.61	1.00	1.12	1.23	1.17	1.28
	SMT-LIB2	1.00	1.05	1.28	1.41	1.57	1.00	1.23	1.32	1.41	1.48
	Combined	1.00	1.01	1.27	1.55	1.54	1.00	1.05	1.24	1.28	1.27
RC	Handcrafted(IR)	1.00	1.02	1.30	1.43	1.58	1.00	1.13	1.76	1.32	1.31
	IR2Vec(IR)	1.00	1.05	1.27	1.42	1.60	1.00	1.19	1.56	2.34	2.22
	SMT-LIB2	1.00	1.07	1.29	1.42	1.66	1.00	1.24	1.32	1.32	1.51
	Combined	1.00	1.08	1.25	1.51	1.69	1.00	1.08	1.23	1.25	1.16
MLP	Handcrafted(IR)	1.00	1.05	1.31	1.27	1.54	1.00	1.17	1.72	1.24	1.36
	IR2Vec(IR)	1.00	1.05	1.25	1.37	1.52	1.00	1.16	1.34	1.36	1.34
	SMT-LIB2	1.00	1.01	1.13	1.31	1.87	1.00	1.22	1.35	1.38	1.93
	Combined	1.00	1.08	1.37	1.49	1.54	1.00	1.08	1.35	1.15	1.15
RF	Handcrafted(IR)	1.00	1.01	1.27	1.22	1.31	1.00	1.09	1.61	1.27	1.30
	IR2Vec(IR)	1.00	1.02	1.41	1.34	1.50	1.00	1.11	1.27	1.23	1.26
	SMT-LIB2	1.00	1.06	1.26	1.39	1.66	1.00	1.21	1.27	1.31	2.19
	Combined	1.00	1.02	1.35	1.27	1.81	1.00	1.07	1.12	1.09	1.01
SMO	Handcrafted(IR)	1.00	1.06	1.23	1.40	1.65	1.00	1.17	1.21	1.20	1.12
	IR2Vec(IR)	1.00	1.06	1.24	1.39	1.56	1.00	1.19	1.20	1.19	1.16
	SMT-LIB2	1.00	1.07	1.28	1.41	1.69	1.00	1.23	1.20	1.18	1.05
	Combined	1.00	1.06	1.27	1.37	1.68	1.00	1.24	1.20	1.16	1.04

In contrast, MLP exhibits unstable and generally suboptimal speedups. For example, on Z3, its performance ranges from 1.01× (1–60s) to 1.54× (300–600s), consistently below DT. This variability suggests overfitting or failure to capture relevant structure for solver selection. RF shows similar limitations: while it achieves a 1.81× speedup on Z3 (300–600s), its performance is less consistent and generally lower than DT, likely due to the averaging effect of its ensemble. kNN and RC perform modestly overall. RC occasionally spikes (e.g., 2.34× on CVC5 in 120–300s), but lacks consistency; kNN yields only marginal gains, indicating limited generalization in the presence of solver diversity.

In summary, these results suggest that simpler models, such as DT, are better aligned with the discrete, hierarchical nature of the pass selection task. The consistent behavior indicates that DT is well-suited for making solver-selection decisions, regardless of problem complexity or the timeout window, likely because it effectively partitions the decision space based on discrete solver behaviors.

Impact of Problem Features. As illustrated in Table 7, Handcrafted and SMT-LIB2 features provided modest yet consistent improvements across most models. For instance, DT with handcrafted IR features achieved a geometric speedup of 1.62× in the 300–600s Z3 range, but improvements in smaller intervals remained marginal (e.g., 1.03× in 1–60s). Similarly, the SMT-LIB2 features offered moderate gains, up to 1.87× for MLP in the hardest Z3 range, but lacked the flexibility to generalize across all ranges or model types.

In contrast, IR2Vec provided notable advantages, especially during time intervals that required a deeper structural grasp. For instance, when paired with IR2Vec, Decision Trees (DT) achieved a 3.28× speedup in the 300–600s range for

Z3 and 3.29× in the 60–120s range for CVC5. The most significant gains emerged from the combination of IR2Vec and DT, which consistently outperformed other pairings in the 60–600s window. This synergy enables lightweight models to scale effectively to more complex problem instances.

Notably, combining IR2Vec with SMT features yielded mixed results. In many cases, the performance gains were similar to or even slightly lower than those with IR2Vec alone. For instance, for DT on CVC5 in the 60–120s range, IR2Vec scored 3.29× while the combined features dropped to 1.06×. This suggests possible feature redundancy or joint-space interference, potentially complicating the decision boundaries. Rather than providing complementary information, the combined features may introduce overlap, thereby impairing learning.

Solver-Specific Performance Trends. The performance of models and features varies across the two solvers. For Z3, speedups remain relatively promising across models and feature combinations, with the highest speedup on the full dataset being 3.28× for DT+IR2Vec in time interval 300-600s. In contrast, CVC5 exhibits greater steadiness, depending on the model and feature set. This may indicate that for Z3, the optimization pass selection is more sensitive to feature quality and model choice. Notably, models such as kNN and RC struggle to achieve competitive speedups on Z3. On the other hand, DT's performance across both solvers and theories highlights its robustness and adaptability. This may be due to DT performing better on training data with little noise, as we carefully refined our training data and excluded unnecessary passes, as discussed in § 6.1.2, which also indicates that IR-level features are highly related to the selection of pass configurations.

6.3 Case Study

To evaluate the generalizability of TUNA-Learn, we conduct a case study using an external dataset derived from the QSF solver [93]. Crucially, these benchmarks were excluded from our training and validation sets (in RQ1 and RQ2), ensuring a fair evaluation on unseen data.

This dataset consists exclusively of 3153 challenging QF_FP (Quantifier-Free Floating Point) constraints. Since floating-point reasoning is a known bottleneck in verification [93], evaluating our model—which was trained on diverse logics—on this specialized, high-complexity dataset helps determine whether it has learned robust, transferable structural patterns rather than overfitting to the training distribution.

We deploy our best-performing model, TUNA-Learn (DT + IR2Vec), in the portfolio setting (Ptf) on the QSF benchmarks and compare its end-to-end solving time against the default solvers (Z3 and CVC5).

The results demonstrate strong generalization capabilities. On this unseen QSF dataset, TUNA-Learn (Ptf) achieves a geometric mean speedup of 1.27× over the default solvers. This performance confirms that the optimization strategies predicted by our model remain effective even when applied to instances from a different distribution and a specialized logic. This provides compelling evidence that the structural features captured by IR2Vec represent intrinsic properties of SMT formulas, making TUNA-Learn a viable approach for optimizing constraints from diverse sources.

6.4 Summary

Our results demonstrate the feasibility and effectiveness of machine learning for selecting optimization passes for SMT simplification. The proposed predictor improves upon both SLOT and default solver configurations. DT combined with IR2Vec embeddings among the evaluated models yields the best performance across various time intervals and solvers. And other models also speed up constraint solving, but show inferior performance. IR2Vec embeddings outperform handcrafted features, indicating their superior ability to capture relevant structural and semantic information. And

SMT-LIB2 features also exhibit suboptimal performance when combined with many ML models, especially on complex constraints, with solving times exceeding 60s. However, augmenting IR2Vec with SMT-LIB2 features surprisingly degrades performance, likely due to redundancy and increased noise in the feature space. While the approach assumes independent prediction of passes and struggles with trivial queries, it nonetheless offers a compelling alternative to expensive iterative search and brittle static configurations. These findings position machine-learned pass selection as a promising direction for future research. In the next section, we outline several avenues for further investigation.

Answer to RQ2: Machine learning methods can effectively predict suitable optimization configurations, with the Decision Tree (DT) model combined with IR2Vec structural embeddings yielding the best performance, especially on complex instances.

7 DISCUSSIONS

This section discusses the limitations of this work and future directions for advancing compiler optimizations-based SMT simplifications.

7.1 Extending the Applicability

We focus on quantifier-free bit-vector and floating-point constraints, following the SLOT approach. There are several directions to expand applicability.

More Variable Types. Our approach builds on SLOT and thus inherits its restriction to bit-vector and floating-point types, a limitation largely stemming from the expressiveness of LLVM IR. Extending support to a broader range of variable types would significantly enhance the applicability of our technique. One promising direction is to perform constraint reduction before translation from SMT-LIB2 to LLVM IR. For instance, Ackermann's reduction [1] can eliminate uninterpreted functions, enabling support for theories such as QF_UFBV. Additionally, array constraints can be reduced to uninterpreted functions using the transformations proposed by Wintersteiger et al. [88], thereby enabling support for theories such as QF_ABV and QF_AUFBV.[1]. These reductions allow us to sidestep the limitations of LLVM IR while broadening the range of supported SMT theories.

Simplifying Quantifiers. While we focus on quantifier-free formulas, quantified formulas play a vital role in applications like invariant inference and program synthesis [42]. Indeed, SMT solvers simplify quantifier reasoning, such as miniscoping [44]. One way to handle quantifiers is to implement simplifications similar to those the compiler passes. Another is to apply lightweight quantifier elimination [15] before using our approach.

Online Simplifications. Currently, we focus on "offline" simplifications, meaning we simplify constraints before invoking the main solving algorithm. Extending our approach to support "online" simplifications [39, 54] could improve the effectiveness. For example, in addition to applying simplifications before the CDCL(T) search, modern solvers often apply simplifications after each restart during the search. However, online simplifications would require translating between the SMT-LIB2 and LLVM IR representations multiple times, which can bring unnoticeable overhead.

Incremental Simplifications. In many applications, SMT solvers are employed in an incremental solving mode, where constraints are added or removed over time, and multiple solver checks are performed. In such cases, incremental simplifications [10] could improve performance by simplifying constraints before each solver check, avoiding repeated simplification of similar constraints. To enable incremental simplifications, an interesting direction is to combine

[1]The tactic "bvarray2uf" in Z3 is based on this algorithm in [88]

incremental compilation [68] and our approach. Indeed, several systems have explored caching mechanisms to reduce solver workload in symbolic execution. KLEE [17] caches branches and counterexamples for bit-vectors and arrays, while Green [82] applies simplification rules to cache results for linear integer arithmetic. Recal [5] canonicalizes formulas using matrix representations to facilitate reuse. GreenTrie [41] extends Green by identifying logical implications between formulas, enabling more aggressive cache hits. Utopia [4] further distinguishes between satisfiable and unsatisfiable queries, applying Sat-delta and Unsat-footprint heuristics to guide reuse. We believe combining client-side incremental mechanisms, such as caching, canonicalization, and dependency tracking, with solver-side incremental simplification can help yield more performance improvements.

7.2 Improving the ML-based Prediction

Our evaluation shows that simple machine learning models with a limited number of optimization flags can achieve high prediction accuracy and significantly improve simplification effectiveness. There are several directions for further purchasing benefits from the approach.

SMT Benchmarks. Our evaluation uses the standard benchmarks from SMT-COMP, drawn from sources such as dynamic symbolic execution (e.g., SAGE), program synthesis, and extended static checking. However, they may not fully capture the diversity of constraints encountered in SMT solver applications. Future work should apply the methodology to a broader range of datasets.

Search Space. After optimizations are selected, the order in which they are applied can significantly affect the final performance, a phenomenon known as the phase ordering problem. For example, applying loop unrolling before dead code elimination may expose additional optimization opportunities, whereas using them in reverse order might miss them. Currently, our approach does not consider the ordering or repetition of passes, similar to the limitations in [3, 19, 99, 100]. Addressing the phase ordering of the optimization passes is stunningly challenging. We may add ranking information to labels to learn the interdependence among different passes, potentially uncovering additional optimization opportunities.

Problem Features. The meta-features used in our current framework capture only a restricted subset of input characteristics. Expanding this feature set offers a promising direction for improving prediction accuracy. In particular, incorporating structural and semantic features directly extracted from SMT-LIB2 constraints may enhance optimization selection strategies [47, 53]. A more ambitious extension involves integrating upstream client information—such as symbolic execution traces or path conditions —that could expose higher-level context about the formula's origin and structure. This additional information may enable more informed and adaptive optimization decisions.

Learning Algorithms. We evaluate several models, including Decision Trees (DT), k-nearest Neighbors (KNN), Ridge Classifiers (RC), Multi-Layer Perceptrons (MLP), and Random Forests (RF). However, future work could further enhance prediction accuracy and explore more advanced program representations [26, 83] and machine learning algorithms.

7.3 Handling Trivial Queries

As noted in Section 4, we do not filter out trivial cases. However, in practice, SMT solvers can resolve certain queries in microseconds. For such trivial constraints, the overhead introduced by translation and optimization may constitute a disproportionate share of the total solving time, surpassing the baseline solving time.

One approach to mitigate this latency is portfolio methodology, which executes a fast, general-purpose solver concurrently with the selected solver. The fast solver may resolve trivial queries before the solver selection overhead

becomes significant, reducing its impact on the total response time. Following SLOT, we apply the portfolio method to TUNA-Learn, achieving a geometric mean speedup of 2.26× and 1.60× over Z3 and CVC5, respectively, when solving nontrivial constraints.

An alternative strategy is to use the fast solver as a pre-solver, invoked before the main solver, akin to the method employed by SATZilla [92]. For example, Boolector [59] initially applies an incomplete local search solver before transitioning to its complete bit-blasting solver. This layered approach enables the system to quickly resolve simple cases while preserving the capability of a more comprehensive solver for handling complex queries.

7.4 Correctness of the Compiler-based Simplifications

We have evaluated the effectiveness of applying compiler optimizations to LLVM IRs derived from SMT-LIB2 formulas. While such transformations can simplify the IR and improve overall SMT-solving performance, they may also introduce subtle errors due to the inherent limitations of the compiler infrastructure and the complex interactions between program transformations and SMT-level semantics. First, LLVM's optimizer is known to contain bugs, and applying optimization passes may trigger latent correctness issues. For example, prior work in compiler testing has demonstrated that varying compiler flags can expose such defects [89, 98]. We have used differential testing to cross-check the configuration results. Although we do not find bugs (in part due to the stable, widely tested SMT-LIB2 benchmarks), compiler transformations may introduce fragile bugs. Second, even without compiler bugs, the transformed IR may violate assumptions made by downstream tools. These tools may rely on specific structural properties of the IR (e.g., SSA or a reducible flow graph) or lack support for certain instructions, leading to crashes, incorrect results, or undefined behavior. For example, if an optimization introduces vector instructions, the translator of SLOT may fail. We can use scalarization (e.g., via the `-scalarize` flag) to preserve correctness in such cases.

8 RELATED WORK

Our work is related to a long line of prior research on constraint solving and compiler optimizations. This section discusses related efforts in addition to the work on SMT simplifications summarized in § 2.

8.1 Constraint Solving

Synthesizing Rewriting Rules for SMT. There has been significant interest in automating the generation of rewriting rules. Nötzli et al. [61] propose a semi-automated approach for enumerating rewrite rules of SMT solvers. SWAPPER [73] combines machine learning with constraint-based synthesis for automatic formula simplification. It identifies candidate rewriting rules by analyzing a corpus of formulas using machine learning. Then, it synthesizes the right-hand side of a rule by first enumerating expressions from a predicate language and applying Sketch [74] to complete the synthesis. Romano and Engler [66] introduces a technique for inferring reduction rules to simplify expressions before passing them to an SMT solver. Other works [58, 60] have also proposed algorithms for automatically generating bit-vector rewriting rules. For example, Nadel [58] employs a combination of constant propagation and equivalence propagation to accelerate bit-vector rewriting across various solvers. Previous works mainly focus on generating reusable rewriting rules offline. These rules are then hard-coded into the solver implementations and used online. Our work does not learn explicit rules but instead selectively applies instance-specific simplification strategies (in the form of compiler optimization passes).

Data-Driven Constraint Solving. Data-driven techniques have been pursued in several ways to speed up constraint solvers. First, the most common approach is algorithm selection [43, 69, 92], which aims to predict the best solver or solver configuration for a given formula. Second, by leveraging Z3's tactic language, recent efforts have formulated tactic optimization as a program synthesis problem. FastSMT [9] synthesizes a solving strategy as a loop-free program with branches. Chen et al. [20] synthesize tailored solving strategies for symbolic execution via a two-stage process that combines offline-trained models and online tuning. Lu et al. [52] propose a Monte Carlo Tree Search-based method that automatically synthesizes an effective strategy. Finally, search-based techniques have been used to solve constraints directly, as opposed to guiding the existing solving algorithms, such as particle-swarm optimization [75], gradient-based search [12, 71], ant colony optimization [78], random walk [31], evolutionary search [50], Monte Carlo Markov Chain (MCMC) [34], fuzzing [12, 50], and classification-based techniques [48].

8.2 Compiler Optimizations

Code Transformations for Program Analysis. Program transformations have been widely studied as a means to improve the precision and efficiency of program analyses, including abstract interpretation [6, 28, 36], invariant generation [8, 70], binary similarity analysis [49, 64], and third-party library detection [90]. The most closely related line of work leverages compiler optimizations to accelerate symbolic execution [16, 18, 96]. Dong et al.[32] empirically analyze the effect of standard compiler optimizations on symbolic execution. Chen et al.[18] further propose a machine learning approach to select optimization sequences that improve symbolic execution.

Our work is closely related to LEO [18] in that both aim to improve constraint solving and involve compile optimizations. However, our method differs in several important respects. First, LEO applies optimizations to the intermediate representation (IR) prior to symbolic execution, thereby influencing not only constraint solving but also other aspects, such as search strategies (e.g., the number of program paths can change). In comparison, we focus exclusively on improving constraint-solving performance. Second, LEO operates on IRs derived from complete C programs and includes specialized mechanisms for handling both application and library code. In contrast, our method targets IRs generated from SMT-LIB formulas, which may originate from a range of sources beyond symbolic execution. These IRs exhibit substantial structural diversity, posing unique challenges for learning-based optimization. Third, we exploit the structural regularity of our inputs—typically loop-free, single-function IRs—by designing a learning approach that combines IR2Vec and new features tailored to this domain. Finally, we discuss broader implications and actionable suggestions for compiler-inspired SMT simplifications, including how to improve their applicability and effectiveness and how to handle trivial queries.

Iterative Auto-Tuning for Compilers. There is a large body of work on compiler auto-tuning, which includes flag selection (choosing the set of optimization passes) and phase ordering (deciding the order in which the passes are applied). Stephenson et al. [77] apply genetic algorithms to tune heuristic priority functions for three compiler optimization passes. OpenTuner [3] autotunes a program using an AUC-Bandit-meta-technique-directed ensemble selection of algorithms. Wang et al. [84] proposes a general auto-tuning system learning dynamic features to tune the reinforcement learning framework. In addition to using iterative search, some works have applied machine learning to guide the search process based on the program's characteristics. For example, in [46], the problem of pass selection is formulated as a Markov process, and supervised learning was used to predict the subsequent optimization based on the current program state. Liu et al. [51] presents a machine-learning-based, iterative compilation optimization method

that uses metric learning and collaborative filtering to recommend passes for the target program. In comparison, our work is the first study to examine the effectiveness of iterative auto-tuning in compiler-based SMT simplifications.

Machine Learning for Compilers. Recent advancements in machine learning (ML) have introduced predictive modeling as a promising approach for non-search-based compiler optimization. In this paradigm, the compiler learns an optimization heuristic offline, replacing traditional hand-tuned methods devised by compiler engineers. With the demonstrated success of ML models across various domains, there is increasing interest in leveraging these techniques to enhance compiler optimization heuristics [25]. Several ML and reinforcement learning (RL) techniques have been proposed to improve a wide range of optimizations, including vectorization [56], loop unrolling and distribution [40], function inlining [79], and register allocation [29]. More recently, large language models (LLMs) have emerged as powerful tools for compiler-related tasks, such as code optimization [27], decompilation [86], and understanding intermediate representations [95]. We refer the readers to [2, 87] for a more comprehensive survey. Our work applies machine learning to determine the optimization passes for IR generated from SMT-LIB2 formulas. Although the translation process may result in some loss of structural information, we observe that features derived from IR2Vec provide substantial end-to-end performance improvements for this task.

9 CONCLUSION

Compiler optimizations have demonstrated significant potential to improve the effectiveness of SMT simplifications, yet their interplay remains insufficiently understood and underexploited. This paper systematically examines the interplay by focusing on two key questions: how different combinations of optimizations influence solver performance and how to predict effective optimization configurations. Our results highlight the potential of iterative compiler optimizations to enhance SMT simplifications and provide strong evidence for using machine learning-driven approaches to automate the selection of optimization passes for SMT solvers.

ACKNOWLEDGEMENTS

We would like to thank the reviewers for their helpful feedback. This work is supported by the National Cryptologic Science Fund of China (2025NCSF02047), National Natural Science Foundation of China (62302434 and U2341212) and CCF-Huawei Populus Euphratica Fund (CCF-HuaweiFM2025004).

REFERENCES

[1] Wilhelm Ackermann and Friedrich Wilhelm Ackermann. 1954. Solvable cases of the decision problem. (1954).
[2] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 81.
[3] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*. ACM, 303–316.
[4] Antonio Aquino, Marc Denecker, and Broes De Cat. 2019. Utopia: SMT-based Formal Analysis Made Easy. In *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings*. IEEE Press, 167–170.
[5] Antonio Aquino, Gereon Inhester, Peter Schrammel, and Yoshinori Yamagata. 2015. Recall: Reuse of Constraint Analysis with Lower Level. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. ACM, 1654–1657.
[6] Vincenzo Arceri, Greta Dolcetti, and Enea Zaffanella. 2023. Unconstrained Variable Oracles for Faster Numeric Static Analyses. In *Static Analysis*, Manuel V. Hermenegildo and José F. Morales (Eds.). Springer Nature Switzerland, Cham, 65–83.
[7] Domagoj Babic. 2008. *Exploiting structure for scalable software verification.* Ph. D. Dissertation.
[8] Gogul Balakrishnan, Sriram Sankaranarayanan, Franjo Ivančić, and Aarti Gupta. 2009. Refining the control structure of loops using static analysis. In *Proceedings of the Seventh ACM International Conference on Embedded Software* (Grenoble, France) *(EMSOFT '09)*. Association for Computing Machinery, New York, NY, USA, 49–58. https://doi.org/10.1145/1629335.1629343

[9] Mislav Balunovic, Pavol Bielik, and Martin Vechev. 2018. Learning to solve SMT formulas. In *Advances in Neural Information Processing Systems*. 10317–10328.

[10] Nikolaj S. Bjørner and Katalin Fazekas. 2023. On Incremental Pre-processing for SMT. In *Automated Deduction - CADE 29 - 29th International Conference on Automated Deduction, Rome, Italy, July 1-4, 2023, Proceedings (Lecture Notes in Computer Science, Vol. 14132)*, Brigitte Pientka and Cesare Tinelli (Eds.). Springer, 41–60. https://doi.org/10.1007/978-3-031-38499-8_3

[11] Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Thorsten Holz. 2017. Syntia: Synthesizing the Semantics of Obfuscated Code. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, Engin Kirda and Thomas Ristenpart (Eds.). USENIX Association, 643–659.

[12] Luca Borzacchiello, Emilio Coppa, and Camil Demetrescu. 2021. Fuzzing Symbolic Expressions. In *Proceedings of the 43rd International Conference on Software Engineering, ICSE 2021*.

[13] Malik Bouchet, Byron Cook, Bryant Cutler, Anna Druzkina, Andrew Gacek, Liana Hadarean, Ranjit Jhala, Brad Marshall, Daniel Peebles, Neha Rungta, Cole Schlesinger, Chriss Stephens, Carsten Varming, and Andy Warfield. 2020. Block public access: trust safety verification of access control policies. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 281–291. https://doi.org/10.1145/3368089.3409728

[14] Robert Brummayer. 2009. *Efficient SMT Solving for Bit-Vectors and the Extensional Theory of Arrays*. Ph. D. Dissertation. Informatik, Johannes Kepler University.

[15] OLIVER BUKOR. [n. d.]. Fast Approximations of Quantifier Elimination for Q3B. ([n. d.]).

[16] Cristian Cadar. 2015. Targeted Program Transformations for Symbolic Execution. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 906–909. https://doi.org/10.1145/2786805.2803205

[17] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, 209–224.

[18] Junjie Chen, Wenxiang Hu, Lingming Zhang, Dan Hao, Sarfraz Khurshid, and Lu Zhang. 2018. Learning to Accelerate Symbolic Execution via Code Transformation. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 109)*, Todd Millstein (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 6:1–6:27. https://doi.org/10.4230/LIPIcs.ECOOP.2018.6

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

[20] Zhenbang Chen, Zehua Chen, Ziqi Shuai, Guofeng Zhang, Weiyu Pan, Yufeng Zhang, and Ji Wang. 2021. Synthesize solving strategy for symbolic execution. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 348–360.

[21] Maciej J. Ciesielski, Tiankai Su, Atif Yasin, and Cunxi Yu. 2020. Understanding Algebraic Rewriting for Arithmetic Circuit Verification: A Bit-Flow Model. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 39, 6 (2020), 1346–1357. https://doi.org/10.1109/TCAD.2019.2912944

[22] Alessandro Cimatti, Alberto Griggio, Ahmed Irfan, Marco Roveri, and Roberto Sebastiani. 2018. Experimenting on solving nonlinear integer arithmetic with incremental linearization. In *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 383–398.

[23] Alessandro Cimatti, Alberto Griggio, Ahmed Irfan, Marco Roveri, and Roberto Sebastiani. 2018. Incremental linearization for satisfiability and verification modulo nonlinear arithmetic and transcendental functions. *ACM Transactions on Computational Logic (TOCL)* 19, 3 (2018), 1–52.

[24] Byron Cook. 2018. Formal Reasoning About the Security of Amazon Web Services. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10981)*, Hana Chockler and Georg Weissenbacher (Eds.). Springer, 38–47. https://doi.org/10.1007/978-3-319-96145-3_3

[25] Keith D. Cooper, Devika Subramanian, and Linda Torczon. 2002. Adaptive Optimizing Compilers for the 21st Century. *J. Supercomput.* 23, 1 (2002), 7–22. https://doi.org/10.1023/A:1015729001611

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

[27] Chris Cummins, Volker Seeker, Dejan Grubisic, Baptiste Roziere, Jonas Gehring, Gabriel Synnaeve, and Hugh Leather. 2025. LLM Compiler: Foundation Language Models for Compiler Optimization. In *Proceedings of the 34th ACM SIGPLAN International Conference on Compiler Construction* (Las Vegas, NV, USA) *(CC '25)*. Association for Computing Machinery, New York, NY, USA, 141–153. https://doi.org/10.1145/3708493.3712691

[28] John Cyphert, Jason Breck, Zachary Kincaid, and Thomas Reps. 2019. Refinement of path expressions for static analysis. *Proc. ACM Program. Lang.* 3, POPL, Article 45 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290358

[29] D Das, S A Ahmad, and V Kumar. 2020. Deep Learning-based Approximate Graph-Coloring Algorithm for Register Allocation. In *Workshop on the LLVM Compiler Infrastructure in HPC*. 23–32. https://doi.org/10.1109/LLVMHPCHiPar51896.2020.00008

[30] Isil Dillig, Thomas Dillig, and Alex Aiken. 2010. Small formulas for large programs: On-line constraint simplification in scalable static analysis. In *International Static Analysis Symposium*. Springer, 236–252.

[31] Peter Dinges and Gul Agha. 2014. Solving complex path conditions through heuristic search on induced polytopes. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 425–436.

[32] Shiyu Dong, Oswaldo Olivo, Lingming Zhang, and Sarfraz Khurshid. 2015. Studying the Influence of Standard Compiler Optimizations on Symbolic Execution. In *Proceedings of the 2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE) (ISSRE '15)*. IEEE Computer

Society, USA, 205–215. https://doi.org/10.1109/ISSRE.2015.7381814

[33] Anders Franzén. 2010. *Efficient solving of the satisfiability modulo bit-vectors problem and some extensions to SMT*. Ph. D. Dissertation. University of Trento.

[34] Zhoulai Fu and Zhendong Su. 2016. XSat: A fast floating-point satisfiability solver. In *International Conference on Computer Aided Verification*. Springer, 187–209.

[35] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA) *(PLDI '05)*. ACM, New York, NY, USA, 213–223. https://doi.org/10.1145/1065010.1065036

[36] Sumit Gulwani, Sagar Jain, and Eric Koskinen. 2009. Control-flow refinement and progress invariants for bound analysis. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. Association for Computing Machinery, New York, NY, USA, 375–385. https://doi.org/10.1145/1542476.1542518

[37] Heqing Huang, Peisen Yao, Rongxin Wu, Qingkai Shi, and Charles Zhang. 2020. Pangolin: Incremental Hybrid Fuzzing with Polyhedral Path Abstraction. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 1613–1627. https://doi.org/10.1109/SP40000.2020.00063

[38] Jeff Huang, Charles Zhang, and Julian Dolby. 2013. CLAP: Recording Local Executions to Reproduce Concurrency Failures. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 141–152. https://doi.org/10.1145/2491956.2462167

[39] Antti EJ Hyvärinen, Matteo Marescotti, and Natasha Sharygina. 2021. Lookahead in partitioning SMT. In *2021 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 271–279.

[40] Shalini Jain, S. VenkataKeerthy, Rohit Aggarwal, Tharun Kumar Dangeti, Dibyendu Das, and Ramakrishna Upadrasta. 2022. Reinforcement Learning assisted Loop Distribution for Locality and Vectorization. In *2022 IEEE/ACM Eighth Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*. 1–12. https://doi.org/10.1109/LLVM-HPC56686.2022.00006

[41] Xiangyang Jia, Carlo Ghezzi, and Shi Ying. 2015. GreenTrie: Efficient Constraint Solving for Software Symbolic Execution. In *Proceedings of the 2015 IEEE International Conference on Software Quality, Reliability and Security*. IEEE, 311–320.

[42] Martin Jonáš and Jan Strejček. [n. d.]. On simplification of formulas with unconstrained variables and quantifiers. In *International Conference on Theory and Applications of Satisfiability Testing (SAT'17)*.

[43] Serdar Kadioglu, Yuri Malitsky, Meinolf Sellmann, and Kevin Tierney. 2010. ISAC-Instance-Specific Algorithm Configuration.. In *ECAI*, Vol. 215. Citeseer, 751–756.

[44] Konstantin Korovin and Margus Veanes. 2014. Skolemization modulo theories. In *International Congress on Mathematical Software*. Springer, 303–306.

[45] Elias Kuiter, Sebastian Krieter, Chico Sundermann, Thomas Thüm, and Gunter Saake. 2022. Tseitin or not Tseitin? The Impact of CNF Transformations on Feature-Model Analyses. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 110:1–110:13. https://doi.org/10.1145/3551349.3556938

[46] Sameer Kulkarni and John Cavazos. 2012. 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 (OOPSLA '12)*.

[47] Will Leeson, Matthew B Dwyer, and Antonio Filieri. 2023. Sibyl: improving software engineering tools with SMT selection. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2185–2197.

[48] Xin Li, Yongjuan Liang, Hong Qian, Yi-Qi Hu, Lei Bu, Yang Yu, Xin Chen, and Xuandong Li. 2016. Symbolic execution of complex program driven by machine learning based constraint solving. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 554–559.

[49] Zongjie Li, Pingchuan Ma, Huaijin Wang, Shuai Wang, Qiyi Tang, Sen Nie, and Shi Wu. 2022. Unleashing the power of compiler intermediate representation to enhance neural program embeddings. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2253–2265. https://doi.org/10.1145/3510003.3510217

[50] Daniel Liew, Cristian Cadar, Alastair F Donaldson, and J Ryan Stinnett. 2019. Just fuzz it: solving floating-point constraints using coverage-guided fuzzing. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 521–532.

[51] Hongzhi Liu, Jie Luo, Ying Li, and Zhonghai Wu. 2021. Iterative Compilation Optimization Based on Metric Learning and Collaborative Filtering. *ACM Trans. Archit. Code Optim.* 19, 1, Article 2, 25 pages. https://doi.org/10.1145/3480250

[52] Zhengyang Lu, Stefan Siemer, Piyush Jha, Joel Day, Florin Manea, and Vijay Ganesh. 2024. Layered and Staged Monte Carlo Tree Search for SMT Strategy Synthesis. *arXiv preprint arXiv:2401.17159* (2024).

[53] Sicheng Luo, Hui Xu, Yanxiang Bi, Xin Wang, and Yangfan Zhou. 2021. Boosting symbolic execution via constraint solving time prediction (experience paper). In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 336–347.

[54] Norbert Manthey, Tobias Philipp, and Christoph Wernhard. 2013. Soundness of inprocessing in clause sharing SAT solvers. In *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 22–39.

[55] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. ACM, New York, NY, USA, 691–701.

https://doi.org/10.1145/2884781.2884807

[56] C Mendis, C Yang, Y Pu, S Amarasinghe, and M Carbin. 2019. Compiler Auto-Vectorization with Imitation Learning. In *NeurIPS'19*, Vol. 32. https://proceedings.neurips.cc/paper/2019/file/d1d5923fc822531bbfd9d87d4760914b-Paper.pdf

[57] Benjamin Mikek and Qirun Zhang. 2023. Speeding up SMT Solving via Compiler Optimization. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1177–1189.

[58] Alexander Nadel. 2014. Bit-Vector Rewriting with Automatic Rule Generation. In *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559*. Springer-Verlag, Berlin, Heidelberg, 663–679. https://doi.org/10.1007/978-3-319-08867-9_44

[59] Aina Niemetz, Mathias Preiner, and Armin Biere. 2014 (published 2015). Boolector 2.0 system description. *Journal on Satisfiability, Boolean Modeling and Computation* 9 (2014 (published 2015)), 53–58.

[60] Aina Niemetz, Mathias Preiner, Andrew Reynolds, Clark Barrett, and Cesare Tinelli. 2018. Solving Quantified Bit-Vectors Using Invertibility Conditions. In *Computer Aided Verification*, Hana Chockler and Georg Weissenbacher (Eds.). Springer International Publishing, Cham, 236–255.

[61] Andres Nötzli, Andrew Reynolds, Haniel Barbosa, Aina Niemetz, Mathias Preiner, Clark Barrett, and Cesare Tinelli. 2019. Syntax-Guided Rewrite Rule Enumeration for SMT Solvers. In *Theory and Applications of Satisfiability Testing – SAT 2019*, Mikoláš Janota and Inês Lynce (Eds.). Springer International Publishing, Cham, 279–297.

[62] Stuart Pernsteiner, Calvin Loncaric, Emina Torlak, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Jonathan Jacky. 2016. Investigating Safety of a Radiotherapy Machine Using System Models with Pluggable Checkers. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 9780)*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer, 23–41. https://doi.org/10.1007/978-3-319-41540-6_2

[63] David Mitchel Perry, Andrea Mattavelli, Xiangyu Zhang, and Cristian Cadar. 2017. Accelerating array constraints in symbolic execution. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, Tevfik Bultan and Koushik Sen (Eds.). ACM, 68–78. https://doi.org/10.1145/3092703.3092728

[64] Xiaolei Ren, Michael Ho, Jiang Ming, Yu Lei, and Li Li. 2021. Unleashing the hidden power of compiler optimization on binary code difference: an empirical study. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 142–157. https://doi.org/10.1145/3453483.3454035

[65] Andrew Reynolds, Maverick Woo, Clark Barrett, David Brumley, Tianyi Liang, and Cesare Tinelli. 2017. Scaling up DPLL (T) string solvers using context-dependent simplification. In *International Conference on Computer Aided Verification*. Springer, 453–474.

[66] Anthony Romano and Dawson Engler. 2013. Expression Reduction from Programs in a Symbolic Binary Executor. In *Model Checking Software*, Ezio Bartocci and C. R. Ramakrishnan (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 301–319.

[67] Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1988. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 12–27.

[68] Patrick W. Sathyanathan, Wenlei He, and Ten H. Tzen. 2017. Incremental whole program optimization and compilation. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO 2017, Austin, TX, USA, February 4-8, 2017*, Vijay Janapa Reddi, Aaron Smith, and Lingjia Tang (Eds.). ACM, 221–232. http://dl.acm.org/citation.cfm?id=3049857

[69] Joseph Scott, Aina Niemetz, Mathias Preiner, Saeed Nejati, and Vijay Ganesh. 2020. MachSMT: A Machine Learning-based Algorithm Selector for SMT Solvers. *Tools and Algorithms for the Construction and Analysis of Systems* 12652 (2020), 303.

[70] Rahul Sharma, Isil Dillig, Thomas Dillig, and Alex Aiken. 2011. Simplifying Loop Invariant Generation Using Splitter Predicates. In *Computer Aided Verification*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 703–719.

[71] Shiqi Shen, Shweta Shinde, Soundarya Ramesh, Abhik Roychoudhury, and Prateek Saxena. 2019. Neuro-Symbolic Execution: Augmenting Symbolic Execution with Neural Constraints.. In *NDSS*.

[72] Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. 2021. Path-sensitive sparse analysis without path conditions. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 930–943.

[73] Rohit Singh and Armando Solar-Lezama. 2016. Swapper: A Framework for Automatic Generation of Formula Simplifiers Based on Conditional Rewrite Rules. In *Proceedings of the 16th Conference on Formal Methods in Computer-Aided Design* (Mountain View, California) *(FMCAD '16)*. FMCAD Inc, Austin, Texas, 185–192.

[74] Armando Solar-Lezama. 2008. *Program Synthesis by Sketching*. Ph. D. Dissertation. Berkeley, CA, USA. Advisor(s) Bodik, Rastislav. AAI3353225.

[75] Matheus Souza, Mateus Borges, Marcelo d'Amorim, and Corina S Păsăreanu. [n. d.]. Coral: Solving complex constraints for symbolic pathfinder. In *NASA Formal Methods Symposium (NFM'11)*.

[76] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*, Vol. 16. The Internet Society, 1–16. http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/driller-augmenting-fuzzing-through-selective-symbolic-execution.pdf

[77] Mark Stephenson, Saman Amarasinghe, Martin Martin, and Una-May O'Reilly. 2003. Meta Optimization: Improving Compiler Heuristics with Machine Learning. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI '03)*.

[78] Julian Thomé, Lwin Khin Shar, Domenico Bianculli, and Lionel Briand. 2017. Search-driven string constraint solving for vulnerability detection. In *Software Engineering (ICSE), 2017 IEEE/ACM 39th International Conference on*. IEEE, 198–208.

[79] Mircea Trofin, Yundi Qian, Eugene Brevdo, Zinan Lin, Krzysztof Choromanski, and David Li. 2021. MLGO: a Machine Learning Guided Compiler Optimizations Framework. *CoRR* abs/2101.04808 (2021). arXiv:2101.04808 https://arxiv.org/abs/2101.04808

[80] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon L. Peyton Jones. 2014. Refinement types for Haskell. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, Johan Jeuring and Manuel M. T. Chakravarty (Eds.). ACM, 269–282. https://doi.org/10.1145/2628136.2628161

[81] S VenkataKeerthy, Rohit Aggarwal, Shalini Jain, Maunendra Sankar Desarkar, Ramakrishna Upadrasta, and YN Srikant. 2020. Ir2vec: Llvm ir based scalable program embeddings. *ACM Transactions on Architecture and Code Optimization (TACO)* 17, 4 (2020), 1–27.

[82] Willem Visser, Jaco Geldenhuys, and Matthew B Dwyer. 2012. Green: reducing, reusing and recycling constraints in program analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 58:1–58:11.

[83] Yao Wan, Zhangqian Bi, Yang He, Jianguo Zhang, Hongyu Zhang, Yulei Sui, Guandong Xu, Hai Jin, and Philip Yu. 2024. Deep Learning for Code Intelligence: Survey, Benchmark and Toolkit. *Comput. Surveys* (2024).

[84] Huanting Wang, Zhanyong Tang, Cheng Zhang, Jiaqi Zhao, Chris Cummins, Hugh Leather, and Zheng Wang. 2022. Automating reinforcement learning architecture design for code optimization. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction* (Seoul, South Korea) *(CC 2022)*. Association for Computing Machinery, New York, NY, USA, 129–143. https://doi.org/10.1145/3497776.3517769

[85] Sheng Wang, Yuan Sun, and Zhifeng Bao. 2020. On the efficiency of K-means clustering: evaluation, optimization, and algorithm selection. *Proceedings of the VLDB Endowment* 14, 2 (Oct. 2020), 163–175. https://doi.org/10.14778/3425879.3425887

[86] Yongpan Wang, Xin Xu, Xiaojie Zhu, Xiaodong Gu, and Beijun Shen. 2025. SALT4Decompile: Inferring Source-level Abstract Logic Tree for LLM-Based Binary Decompilation. arXiv:2509.14646 [cs.SE] https://arxiv.org/abs/2509.14646

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

[88] Christoph M. Wintersteiger, Youssef Hamadi, and Leonardo de Moura. 2010. Efficiently solving quantified bit-vector formulas. In *Formal Methods in Computer Aided Design*. 239–246.

[89] Dongwei Xiao, Zhibo Liu, Yiteng Peng, and Shuai Wang. 2025. Mtzk: Testing and exploring bugs in zero-knowledge (zk) compilers. In *NDSS*.

[90] Zifan Xie, Ming Wen, Tinghan Li, Yiding Zhu, Qinsheng Hou, and Hai Jin. 2024. How Does Code Optimization Impact Third-party Library Detection for Android Applications?. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) *(ASE '24)*. Association for Computing Machinery, New York, NY, USA, 1919–1931. https://doi.org/10.1145/3691620.3695554

[91] Dongpeng Xu, Binbin Liu, Weijie Feng, Jiang Ming, Qilong Zheng, Jing Li, and Qiaoyan Yu. 2021. Boosting SMT solver performance on mixed-bitwise-arithmetic expressions. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 651–664. https://doi.org/10.1145/3453483.3454068

[92] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. 2008. SATzilla: Portfolio-Based Algorithm Selection for SAT. *J. Artif. Int. Res.* 32, 1 (June 2008), 565–606.

[93] Xu Yang, Zhenbang Chen, Wei Dong, and Ji Wang. 2025. QSF: Multi-objective Optimization Based Efficient Solving for Floating-Point Constraints. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE024 (June 2025), 22 pages. https://doi.org/10.1145/3715739

[94] Xinyi Zhang, Zhuo Chang, Yang Li, Hong Wu, Jian Tan, Feifei Li, and Bin Cui. 2022. Facilitating Database Tuning with Hyper-Parameter Optimization: A Comprehensive Experimental Evaluation. arXiv:2110.12654 [cs.DB] https://arxiv.org/abs/2110.12654

[95] Yifan Zhang and Kevin Leach. 2025. Training Large Language Models to Comprehend LLVM IR via Feedback-Driven Optimization. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering* (Clarion Hotel Trondheim, Trondheim, Norway) *(FSE Companion '25)*. Association for Computing Machinery, New York, NY, USA, 1477–1478. https://doi.org/10.1145/3696630.3731662

[96] Yue Zhang, Melih Sirlanci, Ruoyu Wang, and Zhiqiang Lin. 2024. When Compiler Optimizations Meet Symbolic Execution: An Empirical Study. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security* (Salt Lake City, UT, USA) *(CCS '24)*. Association for Computing Machinery, New York, NY, USA, 4212–4225. https://doi.org/10.1145/3658644.3670372

[97] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. 2019. Send Hardest Problems My Way: Probabilistic Path Prioritization for Hybrid Fuzzing. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. https://www.ndss-symposium.org/ndss-paper/send-hardest-problems-my-way-probabilistic-path-prioritization-for-hybrid-fuzzing/

[98] Hao Zhong. 2025. Understanding Compiler Bugs in Real Development . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Los Alamitos, CA, USA, 605–605. https://doi.org/10.1109/ICSE55347.2025.00068

[99] Mingxuan Zhu and Dan Hao. 2023. Compiler Auto-Tuning via Critical Flag Selection. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1000–1011.

[100] Mingxuan Zhu, Dan Hao, and Junjie Chen. 2024. Compiler Autotuning through Multiple-phase Learning. *ACM Transactions on Software Engineering and Methodology* 33, 4 (2024), 1–38.

10 APPENDIX

Table 8. Handcrafted features for the LLVM IR translated from SMT-LIB2 instances

0	Number of BB where total args for phi nodes >5	28	Number of And insts
1	Number of BB where total args for phi nodes is [1,5]	29	Number of BB's with instructions between [15,500]
2	Number of BB's with 1 predecessor	30	Number of BB's with less than 15 instructions
3	Number of BB's with 1 predecessor and 1 successor	31	Number of BitCast insts
4	Number of BB's with 1 predecessor and 2 successors	32	Number of Br insts
5	Number of BB's with 1 successor	33	Number of Call insts
6	Number of BB's with 2 predecessors	34	Number of GetElementPtr insts
7	Number of BB's with 2 predecessors and 1 successor	35	Number of ICmp insts
8	Number of BB's with 2 predecessors and successors	36	Number of LShr insts
9	Number of BB's with 2 successors	37	Number of Load insts
10	Number of BB's with >2 predecessors	38	Number of Mul insts
11	Number of BB's with Phi node # in range (0,3]	39	Number of Or insts
12	Number of BB's with more than 3 Phi nodes	40	Number of PHI insts
13	Number of BB's with no Phi nodes	41	Number of Ret insts
14	Number of Phi-nodes at beginning of BB	42	Number of SExt insts
15	Number of branches	43	Number of Select insts
16	Number of calls that return an int	44	Number of Shl insts
17	Number of critical edges	45	Number of Store insts
18	Number of edges	46	Number of Sub insts
19	Number of occurrences of 32-bit integer constants	47	Number of Trunc insts
20	Number of occurrences of 64-bit integer constants	48	Number of Xor insts
21	Number of occurrences of constant 0	49	Number of ZExt insts
22	Number of occurrences of constant 1	50	Number of basic blocks
23	Number of unconditional branches	51	Number of instructions (of all types)
24	Number of Binary operations with a constant operand	52	Number of memory instructions
25	Number of AShr insts	53	Number of non-external functions
26	Number of Add insts	54	Total arguments to Phi nodes
27	Number of Alloca insts	55	Number of Unary operations