LPO: Discovering Missed Peephole Optimizations with Large Language Models

Zhenyang Xu*
Cheriton School of Computer Science, University of Waterloo
Waterloo, Canada
zhenyang.xu@uwaterloo.ca

Hongxu Xu*
Cheriton School of Computer Science, University of Waterloo
Waterloo, Canada
hongxu.xu@uwaterloo.ca

Yongqiang Tian
Department of Software Systems & Cybersecurity, Monash University
Melbourne, Australia
yongqiang.tian@monash.edu

Xintong Zhou
Cheriton School of Computer Science, University of Waterloo
Waterloo, Canada
x27zhou@uwaterloo.ca

Chengnian Sun
Cheriton School of Computer Science, University of Waterloo
Waterloo, Canada
cnsun@uwaterloo.ca

Abstract

Peephole optimization is an essential class of compiler optimizations that targets small, inefficient instruction sequences within programs. By replacing such suboptimal instructions with refined and more optimal sequences, these optimizations not only directly optimize code size and performance, but also enable more transformations in the subsequent optimization pipeline. Despite their importance, discovering new and effective peephole optimizations remains challenging due to the complexity and breadth of instruction sets. Prior approaches either lack scalability or have significant restrictions on the peephole optimizations that they can find.

This paper introduces LPO, a novel automated framework to discover missed peephole optimizations. Our key insight is that, Large Language Models (LLMs) are effective at creative exploration but susceptible to hallucinations; conversely, formal verification techniques provide rigorous guarantees but struggle with creative discovery. By synergistically combining the strengths of LLMs and formal verifiers in a closed-loop feedback mechanism, LPO can effectively discover verified peephole optimizations that were previously missed.

We comprehensively evaluated LPO within LLVM ecosystems. Our evaluation shows that LPO can successfully identify up to 22 out of 25 previously reported missed optimizations in LLVM. In contrast, the recently proposed superoptimizers for LLVM, Souper and Minotaur detected 15 and 3 of them, respectively. More importantly, within eleven months of development and intermittent testing, LPO found 62 missed peephole optimizations, of which 28 were confirmed and an additional 13 had already been fixed in LLVM. These results demonstrate LPO's strong potential to continuously uncover new optimizations as LLMs' reasoning improves.

CCS Concepts: • **Software and its engineering → Compilers**.

Keywords: compiler, peephole optimization, large language model

ACM Reference Format:
Zhenyang Xu, Hongxu Xu, Yongqiang Tian, Xintong Zhou, and Chengnian Sun. 2026. LPO: Discovering Missed Peephole Optimizations with Large Language Models. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '26), March 22–26, 2026, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3779212.3790184

1 Introduction

Compiler optimizations are a cornerstone of modern software development, designed to enhance a program's performance without altering its behavior [6]. A large portion of optimizations are applied in a series of modular stages called compiler optimization passes, which analyze and transform a program's intermediate representation (IR). Among these, peephole optimization is a critical technique that performs local transformations on small, suboptimal instruction sequences. It operates by identifying a "window" (or "peephole") of dependent instructions and replacing them with a potentially more efficient and refined (i.e., equivalent or with less non-determinism when the original sequence has undefined behavior) sequence. This process is essential for tasks like eliminating redundant code, performing constant folding, and simplifying complex operations. Importantly, these local improvements can also enable more significant

*Zhenyang Xu and Hongxu Xu contributed equally to this work.

ASPLOS '26, Pittsburgh, PA, USA

ACM ISBN 979-8-4007-2359-9/2026/03
https://doi.org/10.1145/3779212.3790184

optimizations by subsequent passes, making peephole optimization a foundational and indispensable technique in compiler design.

Despite its importance, the development of a comprehensive, effective peephole optimizer remains a significant challenge. Unlike other optimizations, such as dead code elimination or loop-invariant code motion, which may be implemented with a single, unified algorithm, peephole optimizations are a collection of pattern-matching and rewriting rules. The complexity and vastness of modern instruction sets cause the number of potential optimization patterns to multiply rapidly, making it exceedingly difficult to develop a complete optimizer. This is why optimizers in widely used compilers like LLVM are continuously evolving, with new patterns being generalized or added [2, 29].

Previous approaches for discovering new peephole optimizations can be classified into three categories:

Manual Inspection: A straightforward way to discover new peephole optimization is manually inspecting the generated IR or assembly code to identify suboptimal instruction patterns. This approach has led to the discovery of a considerable number of optimizations; however, it is labor-intensive, requires deep domain expertise, and has limited scalability.

Differential Testing: Prior studies have proposed using differential testing to automatically detect missed optimization opportunities [9, 17, 22, 36]. This is typically performed in one of the two ways: by identifying differences among the results of different optimizing compilers, or by comparing the compilation results of a pair of semantically equivalent programs. While these approaches have proven effective, they are limited to discovering optimizations that have already been implemented in other compilers or that are not performing as expected in certain cases, rather than discovering entirely new classes of missed optimizations.

Superoptimization: Given a sequence of instructions, a superoptimizer aims to find the optimal instruction sequence through stochastic search or program synthesis [7, 8, 10, 18, 21, 34, 35]. The resulting transformation can then be generalized and implemented as a peephole optimization. However, these approaches are typically computationally expensive, which constrains the code size they can process and limits them to supporting only a subset of the instruction set. For example, Souper, a synthesized superoptimizer for LLVM IR, does not support memory, floating-point, or vector instructions.

LPO. This paper explores the feasibility of using Large Language Models (LLMs) to discover new peephole optimizations. While LLMs are highly effective at creative exploration and reasoning, they are prone to generating incorrect or "hallucinated" answers. Conversely, formal verification can guarantee the correctness of an optimization but is often unsuited for the creative task of searching for new ones. We propose LPO, a novel technique that resolves this tension by combining the creative search capabilities of LLMs with the rigorous guarantees of formal verification. The LPO framework operates as an automated discovery engine for missed peephole optimizations. Its workflow is built around three key components: an extractor, an LLM-based optimizer, and a verifier. First, the extractor analyzes programs to identify candidate instruction sequences for optimization. These candidates are then passed to the LLM-based optimizer, which suggests a potentially more efficient equivalent. The verifier then rigorously checks the proposed optimization for refinement and performance improvement. If the optimization is verified, it is saved for further investigation. If the verification fails, the verifier's output is used as targeted feedback to the LLM, guiding the LLM to correct its proposal in a new attempt. This closed-loop process allows us to systematically and automatically find new, verified peephole optimizations.

We thoroughly evaluated LPO's ability to find new peephole optimizations. We first curated a benchmark suite of 25 missed peephole optimizations that were recently reported in the LLVM repository. Our results show that LPO can identify up to 22 of them, while Souper and Minotaur, two recently proposed synthesizing superoptimizers, detected 15 and 3. Furthermore, we tested LPO's capability to discover unreported, missed optimizations. Within eleven months of development and intermittent testing, LPO found 62 missed peephole optimizations, with 28 confirmed and an additional 13 already fixed in LLVM. Notably, Souper and Minotaur fail to identify 26 and 31 of these confirmed or fixed cases, respectively. These results clearly demonstrate the strong potential of our approach, which combines the creative search of LLMs with the rigorous correctness of formal verification, to continuously find new optimizations as the reasoning abilities of LLMs advance.

Contributions. We make the following contributions.

- We present LPO, a novel framework that synergistically combines the creative search capabilities of LLMs with the rigorous guarantees of formal verification, providing an automated solution to the long-standing challenge of discovering new peephole optimizations.
- We introduce a new methodology for compiler optimization discovery that leverages LLMs to overcome the limitations of prior approaches like manual inspection, differential testing, and traditional superoptimization. A key innovation of our method is a closed-loop feedback mechanism, where formal verification results are used to iteratively guide the LLM's search for correct optimizations.
- We comprehensively evaluated LPO in the LLVM ecosystem and showed that it can uncover previously unreported missed optimizations. LPO identified 62 missed peephole optimizations, with 28 confirmed and 13 already fixed in LLVM. Notably, 26 and 31 of these optimizations cannot be detected by Souper and Minotaur, two synthesizing superoptimizers for LLVM, respectively. These results highlight

the advantages of LPO over prior approaches and establish it as a validated, practical tool for compiler development.
- To ensure reproducibility and facilitate future research, we have released the replication package and experimental data at https://github.com/uw-pluverse/lpo-artifact.

2 Background

We built LPO on top of LLVM and used LPO to find missed peephole optimizations within LLVM. We chose LLVM because of its flexible and user-friendly tool-chain. While our implementation targets LLVM, the core concept of LPO is general and can be applied to other compilers. This section introduces necessary background knowledge on LLVM IR, peephole optimizations, and translation validation.

2.1 LLVM Intermediate Representation

LLVM is an open-source compiler framework for various programming languages and architectures [5, 19, 20]. It has a low-level, typed IR in static single-assignment form, designed for expressiveness and extensibility.

LLVM IR programs consist of *modules*. Each module is a translation unit of the input program, and also is the top level container of all other IR objects, such as functions and global variables [3]. Figure 1d illustrates an LLVM IR module (irrelevant instructions, attributes and declarations are hidden for clarity) that contains a function generated from the original Rust function `clamp` in Figure 1a. The IR function is organized into basic blocks, such as `vector.body` (line 4) shown in the figure. Each basic block comprises a sequence of instructions, has a single entry point (optionally denoted by a label), and ends with a terminator instruction that determines which basic block will be executed next, or whether the function will return. (e.g., `br` and `ret` instructions) [3].

2.2 Peephole Optimization in LLVM

Peephole optimizations identify and replace inefficient patterns with more efficient ones, by examining small instruction windows [27]. For instance, the condition $a - b > a + b$ can be optimized to $b < 0$. Modern peephole optimizations are applied not only to assembly code, but also at higher compilation levels, including abstract syntax tree and IR [13].

The `InstCombine` pass in LLVM is the primary peephole optimization pass [29, 30], which applies algebraic simplifications by pattern matching and replacing instruction sequences with more efficient alternatives [24]. `InstCombine` also performs *canonicalization*, transforming instructions into a consistent form. For example, binary operators with constant operands are rewritten to place the constant on the right-hand side. Such canonicalization reduces the pattern search space and facilitates further optimizations.

However, the development of a comprehensive and effective peephole optimizer is extremely challenging [1, 12]. Currently, the LLVM compiler contains over 25,000 lines of C++ code dedicated to peephole optimizations, and this number is still growing with new patterns added or existing patterns being generalized [2, 29].

Missed Optimization. The Rust program in Figure 1a and its corresponding LLVM IR in Figure 1d demonstrate a reported missed peephole optimization. The instruction sequence highlighted in the rectangle is suboptimal and can be further optimized. After analysis and simplification, the missed optimization can be illustrated with a pair of minimized LLVM functions as shown in Figures 1b and 1c. In this example, the `src` function clamps the input using `x < 0 ? 0 : umin(x, 255)`, whereas the `tgt` function achieves the same effect with `umin(smax(x, 0), 255)`. This transformation reduces the instruction count, resulting in a more efficient implementation.

2.3 Superoptimization

Superoptimization aims to transform instruction sequences to their optimal implementation by searching the space of possible instructions [26]. A superoptimizer can work either as an online optimizer that performs optimization during compilation, or as an offline optimization generator that helps identifying missed optimization. As the computational cost of superoptimization grows exponentially with the length of the instruction sequence (i.e., the search space), superoptimization is usually restricted within small code regions and is often used for finding missed peephole optimizations [7, 10, 34].

Souper. Souper [34] is a superoptimizer for LLVM IR to identify more efficient integer-typed instruction sequences by extracting functions from a module and traversing dataflow edges backward from the return instruction. Unlike exhaustive search, Souper employs counterexample-guided synthesis to accelerate the discovery of optimal replacements. However, its applicability is limited, as it does not support memory accesses, floating-point, or vector instructions.

2.4 Translation Validation

Translation validation is a formal technique for verifying that the target code produced by compilers or optimizers correctly implements the source code [23, 31, 32]. An automatic translation validation process requires a formal specification of "correct implementation", which is a refinement relation [32]. For instance, the LLVM IR function produced by a transformation pass, such as the `InstCombine` pass, should display a subset of the behaviors of the original IR function. The transformation can only eliminate non-determinism by refining undefined behaviors, without introducing new ones. When no undefined behavior exists, the refinement relation becomes an equivalence [23].

Alive2. *Alive2* [23] is an automated translation validation tool for LLVM. Given a pair of LLVM functions, Alive2 verifies whether the transformation from the source function

```rust
pub fn clamp(inp: &[i32], out: &mut [u8]) {
    for (&i, o) in inp.iter().zip(out.iter_mut()) {
        *o = i.clamp(0, 255) as u8;
    }
}
```

(a) A `clamp` function that clamps a sequence of signed 32-bit integers to unsigned 8-bit integers.

```
define i8 @src(i32 %0) {
  %2 = icmp slt i32 %0, 0
  %3 = tail call i32 @llvm.umin.i32(i32 %0, i32 255)
  %4 = trunc nuw i32 %3 to i8
  %5 = select i1 %2, i8 0, i8 %4
  ret i8 %5
}
```

(b) The suboptimal instruction sequence.

```
define i8 @tgt(i32 %0) {
  %2 = tail call i32 @llvm.smax.i32(i32 %0, i32 0)
  %3 = tail call i32 @llvm.umin.i32(i32 %2, i32 255)
  %4 = trunc nuw i32 %3 to i8
  ret i8 %4
}
```

(c) The expected optimal instruction sequence.

```
1  ...
2  define void @clamp(...){
3    ...
4  vector.body:
5    %i = phi i64 [0,%vector.ph],[%i.next,%vector.body]
6    %0 = getelementptr inbounds nuw i32,ptr %inp,i64 %i
7    %1 = getelementptr inbounds nuw i8,ptr %out,i64 %i
8    %2 = getelementptr inbounds nuw i8,ptr %0,i64 16
9    %wide.load = load <4 x i32>,ptr %0,align 4
10   %wide.load7 = load <4 x i32>,ptr %2,align 4
11   %3 = icmp slt <4 x i32> %wide.load,zeroinitializer
12   %4 = icmp slt <4 x i32> %wide.load7,zeroinitializer
13   %5 = tail call <4 x i32> @llvm.umin.v4i32(
14     <4 x i32> %wide.load, <4 x i32> splat (i32 255) )
15   %6 = tail call <4 x i32> @llvm.umin.v4i32(
16     <4 x i32> %wide.load7, <4 x i32> splat (i32 255) )
17   %7 = trunc nuw <4 x i32> %5 to <4 x i8>
18   %8 = trunc nuw <4 x i32> %6 to <4 x i8>
19   %9 = select <4 x i1> %3,
20     <4 x i8> zeroinitializer, <4 x i8> %7
21   %10 = select <4 x i1> %4,
22     <4 x i8> zeroinitializer, <4 x i8> %8
23   %11 = getelementptr inbounds nuw i8, ptr %1, i64 4
24   store <4 x i8> %9, ptr %1, align 1
25   store <4 x i8> %10, ptr %11, align 1
26   %i.next = add nuw i64 %i, 8
27   %12 = icmp eq i64 %i.next, %n.vec
28   br i1 %12, label %middle.block, label %vector.body
29 middle.block:
30   ...
31 } ...
```

(d) The LLVM IR module of the `clamp` function in Figure 1a, with the suboptimal instruction sequence highlighted by the dashed box.

Figure 1. (a) and (d) show a Rust function `clamp` revealing a missed optimization in LLVM, and its corresponding LLVM IR module. (b) and (c) are a pair of simplified LLVM IR functions illustrate the essence of the optimization.

to the target function is correct (i.e., whether the source is refined by the target). For example, given the LLVM function pair shown in Figures 1b and 1c, Alive2 can prove that the transformation from `src` to `tgt` is correct. Conversely, if a transformation is proven to be incorrect, Alive2 provides a counterexample that refutes the refinement relation between the source and the target. Alive2 is widely used in the LLVM community and is integrated into the LLVM development workflow [1].

3 Methodology

Figure 2 and Algorithm 1 show the overall workflow of LPO.

Step ①: Extract instruction sequences. Given a corpus of programs in LLVM IR that have been compiled and optimized by LLVM, LPO first traverses the basic blocks in these IR programs and extracts all the dependent instruction sequences within each basic block (lines 3–4). The details about the extraction process is introduced in § 3.2.

Step ②③⑦: Optimize the instruction sequence. For each instruction sequence, LPO prompts an LLM to output the optimal instruction sequence (line 9 and step ②). For the candidate generate by the LLM, LPO runs three checks to verify whether it potentially reveals a valid optimization (lines 10–22). If all checks are passed, the instruction sequence and the corresponding candidate generated by the LLM are saved for further analysis (line 23 and step ⑦).

Step ③④⑤⑥: Verify the candidate and provide feedback. LLMs are not always reliable. The candidate produced by LLMs can, for example, (1) have syntax error, (2) be suboptimal, (3) be not canonical, (4) have no refinement relation to the original instruction sequence, or (5) be not interesting as it is identical to the original sequence or is less efficient. To tackle this limitation of LLMs, LPO incorporates a verification phase as outlined by the dashed line in Figure 2 to check all these mistakes that can be made by the LLM (steps ③, ④, and ⑤). First, the instruction sequence is sent to the LLVM optimizer, `opt`. If there is any syntax error, LPO uses the error message from `opt` as feedback and prompt the LLM to generate a new candidate (lines 11–14). Otherwise, the optimized instruction sequence output by `opt` becomes the new candidate (line 15). Next, LPO checks whether the candidate is interesting, i.e., whether the candidate potentially manifests a beneficial optimization (more details in § 3.3). If the candidate is not interesting (e.g., is identical to the

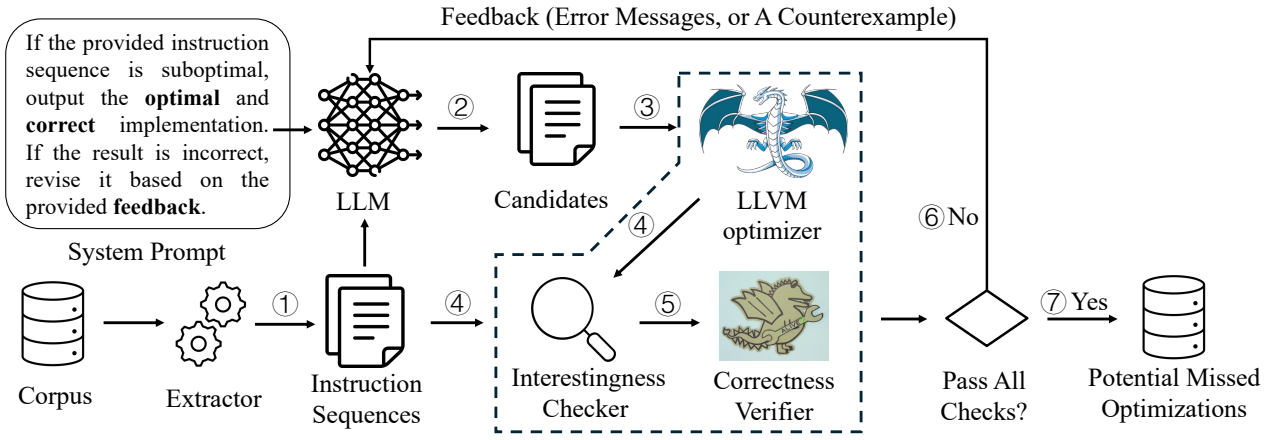

Figure 2. The overall workflow of LPO. The verification phase is outlined by the dashed line.

original instruction sequence), LPO abandons this instruction sequence and moves to the next one (lines 16–17). If the candidate is interesting, LPO further checks the correctness with Alive2 (line 18). If the optimization is proven to be incorrect, LPO uses the counterexample output by Alive2 as the feedback and prompts the LLM to output a correct version based on the feedback (lines 19–22). A parameter ATTEMPT_LIMIT can be configured to specify how many attempts are allowed at most for optimizing each instruction sequence (line 8). In our implementation, this parameter is set to 2.

The remainder of this section details the workflow. § 3.1 illustrates the workflow with a concrete example. § 3.2 elaborates on how LPO extracts instruction sequences from the corpus. § 3.3 details the verification process.

3.1 Illustrative Example

We use the real-world LLVM module in Figure 1d (§ 2) to illustrate the workflow of LPO. Given this module, LPO first traverses all the basic blocks in each function of this module and extracts instruction sequences from them. Figure 3a shows one of the instruction sequences extracted from the basic block `vector.body`.

LPO then provides the extracted instruction sequence to an LLM and prompts it to produce a correct and optimal instruction sequence. In this example, LPO uses gemini-2.0-flash-thinking-exp-01-21 (referred to as Gemini2.0T) as its underlying LLM, and the initial candidate produced by this model, shown in Figure 3b, contains a syntax error.

After obtaining this candidate, LPO initiates the verification process. It first invokes the LLVM optimizer `opt` to check candidate's syntax and to potentially further optimize and canonicalize the candidate. In this case, since the candidate contains a syntax error, `opt` outputs an error message as shown in Figure 3c. LPO then feeds this error message back to the Gemini2.0T, prompting it to produce a new candidate. Figure 3d shows the corrected candidate generated by Gemini2.0T after receiving the feedback.

With this new candidate, LPO again invokes `opt`, and now the syntax check passes. Meanwhile, no further optimization and canonicalization is performed by `opt` since the candidate is already optimal and canonical. The subsequent interestingness checking (discussed in § 3.3) also passes, as the candidate contains one fewer instructions than the original instruction

Algorithm 1: The overall workflow of LPO

```
Input  : corpus, a list of LLVM IR module
Output : A set of instruction sequence pairs that
         potentially reveal missed peephole optimizations
1  instSeqs ← ∅
2  dedup_set ← ∅
3  foreach module ∈ corpus do
4      instSeqs ← instSeqs ∪ Extract(module, dedup_set)
5  foreach instSeq ∈ instSeqs do
6      counter ← 0
7      feedback ← empty string
8      while counter < ATTEMPT_LIMIT do
9          candidate ← InvokeLLM(instSeq, feedback)
10         result_OPT ← RunOPT(candidate)
11         if result_OPT.is_failed then
12             counter ← counter + 1
13             feedback ← result_OPT.error_message
14             continue
15         candidate ← result_OPT.new_candidate
16         if ¬IsInteresting(candidate) then break
17
18         result_Alive2 ← RunAlive2(instSeq, candidate)
19         if ¬result_Alive2.is_correct then
20             counter ← counter + 1
21             feedback ← result_Alive2.counter_example
22             continue
23         save instSeq and candidate for further analysis
24         break
```

```
define <4 x i8> @src(i64 %a0, ptr %a1) {
entry:
 %0 = getelementptr inbounds nuw i32,ptr %a1,i64 %a0
 %wide.load = load <4 x i32>, ptr %0, align 4
 %3 = icmp slt <4 x i32> %wide.load, zeroinitializer
 %5 = tail call <4 x i32> @llvm.umin.v4i32(
   <4 x i32> %wide.load, <4 x i32> splat (i32 255) )
 %7 = trunc nuw <4 x i32> %5 to <4 x i8>
 %9 = select <4 x i1> %3,
   <4 x i8> zeroinitializer, <4 x i8> %7
 ret <4 x i8> %9
}
```

(a) An instruction sequence (wrapped as a function) extracted by LPO from the LLVM IR module shown in Figure 1d. The dashed box highlights the instructions to be optimized.

```
define <4 x i8> @src(i64 %a0, ptr %a1) {
entry:
  %0 = getelementptr inbounds nuw i32,ptr %a1,i64 %a0
  %wide.load = load <4 x i32>, ptr %0, align 4
  %smax_0 = smax <4 x i32> %wide.load, zeroinitializer
  %smin_255 = smin <4 x i32> %smax_0, splat (i32 255)
  %result = trunc nuw <4 x i32> %smin_255 to <4 x i8>
  ret <4 x i8> %result
}
```

(b) The syntactically invalid candidate initially output by Gemini2.0T.

```
error: expected instruction opcode
  %smax_0 = smax <4 x i32> %wide.load, zeroinitializer
            ^
```

(c) The error message output by opt.

```
define <4 x i8> @src(i64 %a0, ptr %a1) {
entry:
  %0 = getelementptr inbounds nuw i32,ptr %a1,i64 %a0
  %wide.load = load <4 x i32>, ptr %0, align 4
  %smax_val = tail call <4 x i32> @llvm.smax.v4i32(
    <4 x i32> %wide.load, <4 x i32> zeroinitializer
  )
  %smin_val = tail call <4 x i32> @llvm.smin.v4i32(
    <4 x i32> %smax_val, <4 x i32> splat (i32 255)
  )
  %result = trunc nuw <4 x i32> %smin_val to <4 x i8>
  ret <4 x i8> %result
}
```

(d) The correct candidate produced by Gemini2.0T after receiving the error message as feedback. The dashed box highlights the new instruction that replaces the original `icmp` and `select` instructions.

Figure 3. An illustrative example demonstrating the workflow of LPO.

sequence. Finally, Alive2 formally confirms the correctness of the proposed optimization, and LPO records the original instruction sequence and the candidate as a potential missed optimization.

Souper cannot detect this missed optimization because it does not support the LLVM intrinsic group `llvm.umin.*`.

3.2 Extracting Instruction Sequences

This subsection introduces the motivation and the design of the extractor (step ①).

Motivation. The motivation for extracting instruction sequences from an entire LLVM IR module is threefold. First, an LLVM IR module can be extremely large and complex, and from our experience, existing LLMs struggle to identify missed peephole optimizations directly from a complete module. By prompting LLMs to focus on optimizing small instruction sequences, we increase the likelihood of discovering potential missed optimizations. Second, extracting instruction sequences from each IR module allows LPO to exclude irrelevant information and deduplicate sequences that appear multiple times in the corpus, and thereby increase the overall efficiency and save the cost for LLM inference. Finally, slicing modules into instruction sequences facilitates subsequent verification and analysis by restricting the scope to a small instruction sequence in each iteration.

Algorithm 2 presents the design of LPO's extractor, whose goal is to obtain all unique dependent instruction sequences from each basic block in every LLVM IR module in the corpus. The extractor takes two inputs: an LLVM IR module and a set, dedup_set containing all previously seen encoded instruction sequences for deduplication. It outputs a set of unique instruction sequences extracted from the given module. Each of the instruction sequences is wrapped as an LLVM function and passed to the LLM to explore potential optimization opportunity within it. The detailed steps are described below.

Extracting From Basic Blocks. Given an LLVM IR module, LPO first initializes result to store the extracted sequences (line 1). It then traverses each basic block bb of each function in the module (lines 2–3), and extracts all dependent instruction sequences contained within bb by calling function `ExtractSeqsFromBB` (line 4). Specifically, `ExtractSeqsFromBB` traverses the instructions in bb in reverse order (line 16). For each instruction inst, if it is a terminator (i.e., the last instruction in a basic block), then it is skipped (line 17), since terminators only define control flow jumps, and LPO currently targets only missed optimization within basic blocks. On line 18, LPO initializes a flag added as false to indicate whether inst has been inserted into any sequence, and initialize a new set, new_set, which is eventually used to update instSeq (line 26). Next, for each instruction sequence instSeq in seq_set, if any instruction in instSeq depends on inst, LPO prepends inst to instSeq and sets added to true (lines 21–23). Otherwise, instSeq remains unchanged (line 24). If inst is not added to any existing sequences—meaning no traversed instruction depends on inst

Algorithm 2: Extracting Instruction Sequences from an LLVM Module – `Extract(module, dedup_set)`

```
   Input  : module, an LLVM module
   Input  : dedup_set, a set that contains all the seen
            encoded instruction sequences
   Output : A set of unique instruction sequences wrapped to
            LLVM IR functions
1  result ← ∅
2  foreach function ∈ module do
3      foreach basic block bb ∈ function do
4          seq_set ← ExtractSeqsFromBB(bb)
5          foreach instSeq ∈ seq_set do
6              wrapped_func ← WrapAsFunc(instSeq)
7              if wrapped_func can be further optimized then
8                  continue
9              digest ← Hash(wrapped_func)
10             if digest ∈ dedup_set then continue
11             dedup_set ← dedup_set ∪ {digest}
12             result ← result ∪ {instSeq}
13 return result
14 Function ExtractSeqsFromBB(bb):
15     seq_set ← ∅
16     foreach instruction inst ∈ bb in reverse order do
17         if inst is a terminator instruction then continue
18         added ← false
19         new_set ← ∅
20         foreach instruction sequence instSeq ∈ seq_set do
21             if any instruction in instSeq depends on inst
                 then
22                 new_set ← new_set ∪ {[inst] + instSeq}
23                 added ← true
24             else new_set ← new_set ∪ {instSeq}
25         if ¬added then new_set ← new_set ∪ {[inst]}
26         seq_set ← new_set
27     return seq_set
```

—LPO creates a new sequence containing only inst and adds the new sequence to new_set (line 25).

Verifying and Deduplicating Instruction Sequences. After LPO traverses all the instructions within a basic block, a set of instruction sequences can be obtained. For each extracted instruction sequence, LPO wraps it as an LLVM IR function (line 6). Specifically, LPO appends a return instruction that returns the value produced by the last instruction to the sequence and treats all undefined operands in the sequence of instructions as function arguments. With the wrapped function obtained, LPO checks that LLVM cannot further optimize the instruction sequence (line 8). This check is necessary because a sequence that is unoptimizable in its original context may become optimizable once isolated. For example, an instruction in the sequence might be used not only by subsequent instructions within the sequence but also by instructions in later basic blocks; once those external uses are removed, the instruction could be combined with others in the sequence. Finally, LPO computes a hash of the function based on the opcode and operands of each instruction (line 9), and add instruction sequences whose hashes have not been recorded in dedup_set (lines 10–12).

3.3 Verifying Outputs of the LLM

After extracting instruction sequences from the corpus and wrapping them as LLVM IR functions (we use the term instruction sequence and function interchangeably in the following), LPO begins prompting the specified LLM to optimize each function. Each LLM-generated candidate is then verified through three steps.

Preprocessing with `opt`. First, LPO feeds the candidate to `opt`, the LLVM optimizer, with the `-O3` flag to perform aggressive optimization (step ③). The step serves two purposes. First, `opt` checks whether the function proposed by the LLM is syntactically correct. If the function contains syntax errors, `opt` outputs an error message, which LPO can then use as feedback to prompt the LLM to fix the error (step ⑥). Second, although the LLM may produce a modified version of the original function, the proposed function may not be canonical or optimal w.r.t. `opt`. In such cases, running `opt` can further optimize and canonicalize the proposed function reducing the effort required for subsequent manual analysis.

Checking Interestingness. If the function proposed by the LLM is syntactically correct, LPO next checks whether the function—after preprocessed by `opt`—potentially manifests a beneficial optimization. Determining whether the transformation from the original function to the proposed one is beneficial is tricky, because (1) comparison between two functions is not straightforward, as multiple aspects can be considered (e.g., code size and performance on a specific target); (2) even if the proposed function appears significantly better, implementing the corresponding optimization might not be beneficial overall if it hinders other, more valuable optimizations. Due to these challenges, this step is designed to only filter out as many uninteresting cases as possible (i.e., those unlikely to manifest a beneficial optimization), thereby reducing the effort required for subsequent manual analysis. In our prototype, two metrics are considered when checking the interestingness, the instruction count and the total cycles measured by `llvm-mca`[1] on a specific target with a specific CPU (e.g., with the target triple set to `x86_64-unknown-unknown` and the CPU set to `btver2`) Functions with fewer instructions or fewer total cycles are considered interesting and thus pass the check. Additionally, functions with the same instruction count and total cycles are also considered interesting if they differ syntactically from the original, since such transformations—though not immediately improving the function—may enable further optimization. Interestingness checking is performed before correctness verification, as it is generally more efficient,

[1] A performance analysis tool to statically measure the performance of machine code in a specific CPU based on LLVM's scheduling models.

avoiding the costlier correctness checking for unpromising cases and thus improving overall workflow efficiency.

Verifying Correctness. If the function proposed by the LLM passes the interestingness checking, LPO then verifies the correctness of the transformation from the original function to the proposed one using Alive2 [23]. If Alive2 proves the transformation incorrect, it provides a counterexample to demonstrate the incorrectness, which LPO uses as feedback to prompt the LLM to propose a new candidate. If Alive2 encounters a fixable error that prevents the proof—for example, mismatched function signatures—the error message is also used as feedback to guide the LLM in avoiding the issue (step ⑥). Otherwise, if Alive2 proves the transformation correct, LPO records the pair of functions as a potential missed optimization (step ⑦)

4 Evaluation

To comprehensively evaluate LPO, we conducted experiments to investigate the following research questions.

RQ1: Can LPO detect prior missed optimizations?

RQ2: Can LPO discover new missed optimizations?

RQ3: How much throughput can LPO achieve, and what is the associated cost?

4.1 Experimental Setup

Selected LLM Models. Table 1 lists all the models selected for evaluating LPO. We selected diverse models to investigate how the capabilities of different LLMs affect the effectiveness of LPO. For RQ1, we selected two open-source and four proprietary models. Specifically, we chose two open-source models with significantly different sizes: Gemma3 (27 billion parameters) and Llama3.3 (70 billion parameters). For proprietary models, we included two base models, i.e., Gemini2.0 and GPT-4.1, and two reasoning models, i.e., Gemini2.0T and o4-mini. These models have relatively early knowledge cut-off dates, allowing us to collect a reasonable number of benchmarks while avoiding potential data leakage. In RQ2, during the long-term, intermittent experiment, we tried several open-source models and used a locally deployed llama3.3:70b for most of the time. In RQ3, Llama3.3 and Gemini2.5 (a cost-efficient and low-latency model recommended for scenarios requiring high throughput) are used to evaluate the throughput and cost, covering two common usage scenarios, i.e., using a locally deployed LLM and invoking an LLM via API. For Gemini2.0T and Gemini2.5, we set the reasoning budget to *low* (i.e., at most 1,024 reasoning tokens).

Baselines. We used three baselines in our evaluations.

- *Souper*. The primary baseline used in our evaluation is Souper [34], the state-of-the-art superoptimizer for LLVM. Souper allows users to set the maximum instructions for enumerative synthesis (referred to as `Enum`). Increasing this value can improve Souper's capability in finding missed optimization but also incurs significantly higher computational cost. We evaluated Souper using different values of `Enum`: the default value 0 ($\text{Souper}_{\texttt{Default}}$) and values from 1 to 3 ($\text{Souper}_{\texttt{Enum}}$). A timeout of 20 minutes is applied to each input to ensure the experiments complete in a reasonable time.
- *Minotaur*. We also compared LPO with Minotaur [21], a recent superoptimizer for LLVM that employs program synthesis techniques and focuses on optimizing integer and floating-point SIMD code. Default settings were used for Minotaur in our experiments.
- *LPO*$^-$. To evaluate the effectiveness of the iterative prompting strategy, we prepared LPO$^-$, a variant of LPO that does not prompt the LLM to generate new candidates using feedback (i.e., it omits steps ④ and ⑦ in Figure 2)

Table 1. The selected LLMs in evaluation. Gemini2.5 is excluded from RQ1 to prevent potential data leakage.

Model Name	Model Version	Reasoning	Cut-off Date
Gemma3	gemma3:27b	No	08/2024
Llama3.3	llama3.3:70b	No	12/2023
Gemini2.0	gemini-2.0-flash	No	08/2024
Gemini2.0T	gemini-2.0-flash-thinking-exp-01-21	Yes	08/2024
GPT-4.1	gpt-4.1-2025-04-14	No	06/2024
o4-mini	o4-mini-2025-04-16	Yes	06/2024
Gemini2.5	gemini-2.5-flash-lite	Yes	01/2025

System Configuration. The experiments were run on an Ubuntu 22.04 server with three NVIDIA RTX 6000 Ada GPUs, Intel Xeon Gold 5217 CPU@3.00GHz, and 384 GB RAM.

4.2 RQ1: Detecting Previously Reported Missed Optimizations

To quantitatively evaluate the effectiveness of LPO in finding missed optimization opportunities and to objectively compare it with the baselines, we conducted a controlled experiment to assess how effectively each tool identifies previously reported missed optimizations. For the benchmark suite, we collected 25 missed peephole optimizations from issues in the official LLVM GitHub repository tagged with both *missed-optimization* and *llvm:instcombine* labels. To avoid potential data leakage [25, 38], we only included issues created after the end of August 2024, which is the latest knowledge cut-off date among all LLMs listed in Table 1 except for Gemini2.5, the model excluded from this research question. For each benchmark, we collected the example suboptimal LLVM IR function provided in the issue and examined whether LPO, LPO$^-$, Souper and Minotaur could detect the missed optimization within the function. To mitigate the effects of the inherent non-determinism of LLMs, we repeated each experiment five times for both LPO and LPO$^-$. The results of the experiment are shown in Table 2.

Effectiveness of LPO with Different LLMs. The effectiveness of LPO heavily depends on the code understanding and

Table 2. Evaluation results on 25 previously reported missed optimizations. For LPO and LPO$^-$, each benchmark is tested with five rounds for each model, and the results show how many times LPO and LPO$^-$ can catch the optimization opportunity. For Souper$_{\texttt{Enum}}$, if it detects the optimization with `Enum` set to any value from 1 to 3, it is marked with a ✓. Empty cells represent that the optimization is not detected by the tool. The numbers of successful times are grouped into three categories: 1-2 time, 3-4 times and all 5 times. The **Average** row shows the average number of successful benchmarks per round for LPO and LPO$^-$, and the **Total** row shows the total number of missed optimizations that are detected at least once.

Issue ID	Gemma3		Llama3.3		Gemini2.0		Gemini2.0T		GPT-4.1		o4-mini		Souper		Minotaur
	LPO$^-$	LPO	LPO$^-$	LPO	LPO$^-$	LPO	LPO$^-$	LPO	LPO$^-$	LPO	LPO$^-$	LPO	`Default`	`Enum`	`Default`
104875							1	5							
107228							5	5		1	4	5		✓	
108451			5	5	5	5	4	5	1	4	2	5		✓	✓
108559			5	5	5	5	4	5	1	4	4	5		✓	
110591			5	5	5	5	2	5	2	5	3	5			
115466	1	1	5	5			5	5	3	4	5	5		✓	
118155							3	3			2	2			
122235				5	2	5		2		5				✓	
122388					5	5	4	4	2	3					
126056						1	5	5	1	4	5	5		✓	✓
128475							2	2			4	5		✓	
128778						1		5		1	3	5		✓	
129947								1							
131444															
131824						1		3		1		3		✓	
132508		2	1	5	5	5	2	2		3	3	5	✓		
134318															
135411							5	5	1	1	5	5		✓	✓
137161						2						1			
141479							5	5			4	5	✓	✓	
141753								1			1	2		✓	
141930	3	3	4	5	2	2	5	5			5	5		✓	
142497								1		1		1			
142593								4				2	✓	✓	
143259															
Average	0.8	1.2	5.2	7	5.8	7.4	10.4	15.6	2.2	7.4	10.0	14.2	N/A	N/A	N/A
Total	2	3	6	7	7	11	14	21	7	12	14	18	3	14	3

reasoning capabilities of the employed LLM, and LLMs with different capabilities can therefore lead to varying levels of effectiveness. As shown in Table 2, the performance of LPO varies significantly across different LLMs. With Gemma3, the smallest and weakest of the selected models, LPO can detect at most 3 out of 25 missed optimizations, and on average only 1.2 per round. With other non-reasoning but more capable models, the performance of LPO improves substantially: using Llama3.3, Gemini2.0, and GPT-4.1, LPO can identify up to 7, 11 and 12 missed optimizations, and on average 7.0, 7.4, and 7.4, respectively. Employing reasoning models further enhance the performance of LPO. With Gemini2.0T and o4-mini, LPO can detect up to 21 and 18 missed optimizations, and on average 15.6 and 14.2, respectively.

LPO vs. Souper and Minotaur. We evaluated Souper with maximum instructions for enumerative synthesis set to 0 (Souper$_{\texttt{Default}}$) and from 1 to 3 (Souper$_{\texttt{Enum}}$). Souper$_{\texttt{Default}}$, designed for efficiency, can identify only 3 out of 25 missed optimizations, while Souper$_{\texttt{Enum}}$, with different values of `Enum`, detected up to 14 out of 25 at the cost of longer execution time. In total, Souper detected 15 missed optimizations (for Issue 132508, Souper$_{\texttt{Enum}}$ either cannot detect or crashed but Souper$_{\texttt{Default}}$ can successfully handle). The most common reason that Souper fails to detect a missed optimization is that the instruction sequence contains instructions unsupported by Souper. Among the 10 missed optimizations that Souper cannot detect, LPO can detect up to 7 of them. In this benchmark, Minotaur detects 3 missed optimizations, all of which are also identified by Souper$_{\texttt{Enum}}$ and by LPO with a reasoning model. These results suggest that, despite its optimization for integer and floating-point SIMD code, Minotaur remains limited in detecting missed peephole optimizations.

LPO vs. LPO$^-$. The iterative prompting strategy improves LPO's effectiveness in identifying missed optimizations regardless of the model employed. Without iterative prompting, LPO$^-$ failed to identify 1 to 7 optimization that LPO can detect across different LLMs. Moreover, the average number of optimizations detected by LPO$^-$ is consistently lower than that of LPO, with the gaps ranging from 0.4 to 5.2.

Table 3. All missed optimizations found by LPO and reported to LLVM. The columns **Souper$_{\texttt{Default}}$**, **Souper$_{\texttt{Enum}}$**, and **Minotaur** indicate whether each tool can detect the corresponding missed optimization. The result "timeout" means Souper$_{\texttt{Enum}}$ cannot detect the missed optimization within a 20-minute timeout with `Enum` set to any value from 1 to 3. Out of 62 issues in total, 28 are confirmed, 13 have been fixed, 4 are duplicates, and 3 are marked as "wontfix".

Issue ID	Status	Souper$_{\texttt{Default}}$	Souper$_{\texttt{Enum}}$	Minotaur
128134	Fixed			
128460	Confirmed		timeout	
130954	Wontfix			
132628	Wontfix		timeout	
133367	Fixed			
139641	Confirmed		✓	✓
139786	Confirmed			
142674	Fixed	✓	✓	
142711	Fixed			
143030	Unconfirmed			
143211	Fixed		✓	
143630	Unconfirmed		✓	
143636	Fixed		✓	
143649	Unconfirmed			
143957	Confirmed	✓	timeout	
144020	Confirmed		✓	✓
152237	Confirmed		✓	✓
152788	Unconfirmed			✓
152797	Confirmed		timeout	
152804	Confirmed	✓	✓	✓
153991	Confirmed			
153999	Duplicate			
154000	Duplicate	✓	✓	
154025	Unconfirmed	✓		
154035	Unconfirmed			
154238	Fixed			
154242	Confirmed			✓
154246	Confirmed			
154258	Unconfirmed	✓	✓	
157315	Fixed			
157370	Fixed		✓	
157371	Fixed			
157372	Duplicate			
157486	Confirmed			
157524	Fixed			
163084	Confirmed		✓	
163093	Unconfirmed			
163108	Fixed		✓	✓
163109	Confirmed			
163110	Confirmed		✓	
163112	Confirmed			
163115	Confirmed			
166878	Confirmed			
166885	Confirmed			
166887	Unconfirmed		✓	✓
166890	Unconfirmed		✓	
166973	Fixed			✓
167003	Confirmed			✓
167014	Confirmed			
167055	Confirmed			
167059	Unconfirmed			
167079	Unconfirmed			
167090	Unconfirmed			✓
167094	Duplicate			
167096	Confirmed			
167173	Confirmed		✓	✓
167178	Unconfirmed		✓	
167183	Confirmed		✓	✓
167190	Confirmed			
167199	Wontfix			
170020	Confirmed		✓	
170071	Confirmed			

4.3 RQ2: Discovering New Missed Optimization

To demonstrate the practical value of LPO, we intermittently ran an experiment that employs LPO to search for new missed optimizations in LLVM. In this experiment, the corpus used by LPO is from an LLVM IR dataset named LLVM Opt Benchmark [37]. This dataset contains optimized IR files from 240 real world projects written in C, C++, or Rust. Since the entire benchmark is too massive to serve as our corpus, we built our corpus by selecting five popular projects for each programming language, i.e., `cpython`, `ffmpeg`, `linux`, `openssl`, `redis`, `node`, `protobuf`, `opencv`, `z3`, `pingora`, `ripgrep`, `typst`, `uv`, `zed`. From this corpus, LPO extracted over 800,000 unique instruction sequences, eliminating around 8.7 million duplicates.

Table 3 lists all missed optimizations found by LPO and reported to LLVM. Within eleven months of development and intermittent testing, LPO has found 62 potential missed peephole optimizations, with 28 confirmed and 13 already fixed in LLVM. It also detected a bug in Alive2.[2] Among all the reported missed optimizations, we marked 3 of them as *wontfix* (i.e., Issue 130954, Issue 132628, and Issue 167199) according to the feedback from the maintainers. The reasons are: (1) the optimization is handled by the backend, (2) the optimization would prevent other valuable optimizations, and (3) the optimization is application-specific and is not general enough.

We also investigated if Souper and Minotaur can detect these reported missed optimizations. As shown in Table 3, Souper$_{\texttt{Default}}$ can identify 6 out of 62 missed optimization, 3 of which are confirmed or fixed; Souper$_{\texttt{Enum}}$ can identify 20, and 14 of them are confirmed or fixed; Minotaur can identify 13, and 10 of them are confirmed or fixed. In the remainder of the section, we present three examples of confirmed missed optimizations found by LPO that neither Souper nor Minotaur can detect.

Case Study 1 (Figures 4a and 4d). The `src` function loads two consecutive 16-bit values from a given address `%0` and combines them into a single 32-bit value using zero extension (`zext`), left shift (`shl`), and an `or` instruction. This function can be optimized to directly loading a 32-bit value from the given address. LLVM originally missed this optimization because it did not support merging consecutive loads with different sizes. Specifically, the original unoptimized function loads three consecutive values—two 8-bit values and one

[2] https://github.com/AliveToolkit/alive2/issues/1229

Table 4. Average case execution time (including timeouts) of LPO (with Llama3.3 and Gemini2.5) vs. Souper$_{\texttt{Default}}$ and Souper$_{\texttt{Enum}}$. We set the timeout to 20 minutes, and the number of timeouts for each tool is also listed.

Tool	LPO		Souper			
	Llama3.3	Gemini2.5	`Default`	`Enum` =1	`Enum` =2	`Enum` =3
Time/Case (s)	26.2	6.7	2.8	37.2	144.4	183.7
# of Timeouts	0	0	0	80	412	616

16-bit value. LLVM successfully merges the two 8-bit loads but fails to further merge them with the subsequent 16-bit load. Souper and Minotaur cannot identify this case, as they cannot synthesize `load` and `getelementptr` instructions.

Case Study 2 (Figures 4b and 4e). In the `src` function, the input `%0` is first clamped to be at least 1 by `llvm.umax.i8`, then doubled with a 1-bit left shift, and finally clamped again to be at least 16. In the `tgt` function, the initial clamp is removed. This simplification is valid because the second clamp to 16 subsumes the effect of the first clamp, which makes it redundant. Souper cannot identify this missed optimization because it does not support the intrinsic family `llvm.umax.*`. Although Minotaur supports synthesizing this operation, it fails to detect the missed optimization.

Case Study 3 (Figures 4c and 4f). In the `src` function, the input `%0` is first checked for being an ordinary (non-NaN) value; if it is NaN, it is replaced with `0.0`, otherwise, `%0` is kept. The result is then compared against `1.0`. In the `tgt` function, the check and replacement are removed, and `%0` is directly compared with `1.0`. This simplification is valid because the instruction `fcmp oeq` (ordered equal) already returns false if `%0` is NaN, so explicitly substituting NaNs with `0.0` is redundant. Souper cannot identify this missed optimization because it does not support floating-point instructions, and Minotaur crashes on this IR function.

4.4 RQ3: Throughput and Cost of LPO

To evaluate the throughput and cost of LPO, we constructed a benchmark suite consisting of 5,000 instruction sequences randomly sampled from those extracted from aforementioned real-world projects. We measured the throughput of LPO under two different setups—using a locally deployed Llama3.3 and using Gemini2.5 via API. For the latter setup, we also recorded the API cost. For comparison, we evaluated Souper$_{\texttt{Default}}$ and Souper$_{\texttt{Enum}}$ on the same benchmark suite as baselines. All tools were executed with a single thread and the results are presented in Table 4.

Overall, the throughput of LPO under both setups was lower than Souper$_{\texttt{Default}}$ but higher than Souper$_{\texttt{Enum=1}}$. Specifically, LPO with a locally deployed Llama3.3 took 36.4 hours in total to process all the cases, averaging 26.2 seconds per case. In contrast, LPO with Gemini2.5 via API achieved a significantly higher throughput, completing the experiment in 9.3 hours with an average of 6.7 seconds per case, at a total cost of approximately 5.4 USD.

For Souper, the default configuration achieved the highest throughput, processing all cases in about 3.9 hours, averaging 2.8 seconds per case. However, as the `Enum` parameter increases, the throughput decreases sharply. With `Enum` set to 1, 2, and 3, the execution time rose to 52 hours, 200 hours, and 255 hours, averaging 37.2, 144.4, and 183.7, respectively. Moreover, Souper failed to terminate within 20 minutes in 80, 412, and 616 cases for `Enum` value of 1, 2, and 3, respectively.

In summary, these results further demonstrate the feasibility and practical value of LPO. Although LPO requires additional computation resources for LLM inference compared to Souper, cost-efficient commercial LLMs (e.g., Gemini2.5 from Google) enables LPO to achieve high throughput with manageable monetary cost. Moreover, as LLMs continue to become faster and more cost-effective, the overall cost-effectiveness of LPO will likewise improve.

4.5 Threats to Validity

The first threat to validity is potential data leakage. When evaluating the effectiveness of LPO in finding previously reported missed optimization, it is possible that the training set of the deployed model already contains information related to the missed optimizations, which could inflate the results. To mitigate this threat, we selected LLMs with clear knowledge cut-off dates and only included missed optimizations reported after those dates in our benchmarks. Another concern is that the collected previously reported missed optimizations may not be fully representative. To mitigate this, we gathered as many benchmarks as possible and manually inspected each case to ensure validity. The non-determinism of LLMs poses a threat to the internal validity. We mitigate this by evaluating LPO with multiple LLMs and repeating the controlled experiment in RQ1 for five times.

5 Discussion

In this section, we discuss the impact of the optimizations discovered by LPO, the key conceptual contribution of the work, and the future work on this research direction.

5.1 Impact of the Discovered Optimizations

In LLVM development practice, each patch implementing a peephole optimization is typically evaluated using LLVM Opt Benchmark [37], a dataset of LLVM IR files collected from 240 real-world projects written in C, C++ or Rust, and the LLVM compile time tracker [33]. Maintainers carefully review each patch and accept it only if: ① the net effect is positive, as demonstrated in the benchmark; and ② the compile-time impact is acceptable. In other words, each fixed missed optimization demonstrates tangible benefits in practice anticipated by the community. To date, 13 reported missed optimizations have been fixed with 15 carefully reviewed

```
define i32 @src(ptr %0) {
  %2 = load i16, ptr %0, align 2
  %3 = getelementptr i8, ptr %0, i64 2
  %4 = load i16, ptr %3, align 1
  %5 = zext i16 %4 to i32
  %6 = shl nuw i32 %5, 16
  %7 = zext i16 %2 to i32
  %8 = or disjoint i32 %6, %7
  ret i32 %8
}
```

(a) Case 1: sub-optimal version.

```
define i8 @src(i8 %0) {
  %2 = call i8 @llvm.umax.i8(
    i8 %0, i8 1)
  %3 = shl nuw i8 %2, 1
  %4 = call i8 @llvm.umax.i8(
    i8 %3, i8 16)
  ret i8 %4
}
```

(b) Case 2: sub-optimal version.

```
define i1 @src(double %0) {
  %2 = fcmp ord double %0, 0.000000e+00
  %3 = select i1 %2,
       double %0, double 0.000000e+00
  %4 = fcmp oeq double %3, 1.000000e+00
  ret i1 %4
}
```

(c) Case 3: sub-optimal version.

```
define i32 @tgt(ptr %0) {
  %2 = load i32, ptr %0, align 2
  ret i32 %2
}
```

(d) Case 1: optimized version.

```
define i8 @tgt(i8 %0) {
  %2 = shl nuw i8 %0, 1
  %3 = call i8 @llvm.umax.i8(
    i8 %2, i8 16)
  ret i8 %3
}
```

(e) Case 2: optimized version.

```
define i1 @tgt(double %0) {
  %2 = fcmp oeq double %0, 1.000000e+00
  ret i1 %2
}
```

(f) Case 3: optimized version.

Figure 4. Three examples of confirmed missed optimizations found by LPO that Souper and Minotaur fail to detect.

patch commits, demonstrating that optimizations discovered by LPO have a real-world impact.

Table 5. The Number of impacted IR files and projects and compile time impact for each accepted patch.

ID	#IR Files	#Projects	Δ in Compile Time (instruction:u)
128134	54	13	+0.02%
133367	122	18	N/A
142674	251	15	+0.05%
142711	10	1	-0.00%
143211	16	4	N/A
143636	2,476	68	+0.02%
154238	10	4	N/A
157315	6	2	+0.00%
157370	N/A	N/A	+0.04%
157371 (1)	10	13	N/A
157371 (2)	28	1	+0.02%
157524	N/A	N/A	-0.03%
163108 (1)	3,055	93	-0.05%
163108 (2)	28	4	-0.01%
166973	759	62	N/A

Prevalence of the Detected Suboptimal Patterns. We analyzed the evaluation results of the patches that fixed the 13 fixed missed optimization using LLVM Opt Benchmark, recording the number of impacted IR files and projects to estimate the prevalence of the corresponding suboptimal patterns in real-world code. As shown in Table 5, most of these handled suboptimal patterns appear in multiple real-world projects. Notably, Issue 143636 affects 2,476 IR files across 68 projects, and Issue 163108 affects over 3,000 files across 97 projects. These results highlight that LPO can identify impactful missed optimizations.

Impacts on the Compile Time. For each patch, we also examined the compile-time impact using the LLVM compiler time tracker. Table 5 reports the geometric mean of the compile-time changes across 10 realistic and representative benchmarks (kimwitu++, sqlite3, consumer-typeset, Bullet, tramp3d-v4, mafft, ClamAV, lencod, SPASS, and 7zip) comparing the compiler before and after the patch. The compile-time metric is measured as `instruction:u`, i.e., the number of machine instructions executed in user space during compilation. A positive change indicates a slowdown. As the table shows, the compile-time impact of each patch is negligible.

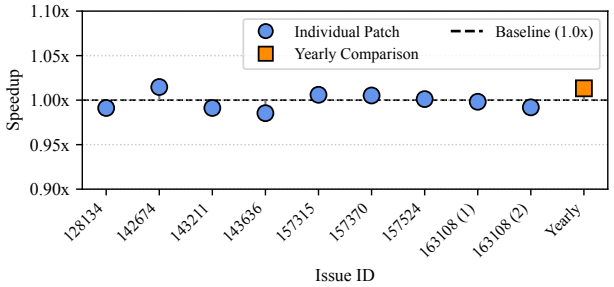

Figure 5. Geometric mean of speedup on SPEC CPU2017 Integer for selected patches, as well as for two LLVM versions (f3501d7 on Dec. 1, 2025 vs. 017c75b on Dec. 1, 2024).

Runtime Efficiency on SPEC CPU2017. To further assess the practical impact of the fixed missed optimizations, we evaluated their runtime effects on the SPEC CPU2017 Integer benchmark suite[3]. We focused on patches that are most

[3]We only included C/C++ benchmarks.

likely to affect these benchmarks.[4] All experiments were conducted on a system equipped with an AMD Ryzen 9 7950X 16-core processor and 128 GB of RAM. Figure 5 reports the geometric mean speedup. Following the instructions of the benchmark suite, each benchmark was executed three times, and the median runtime was used for comparison [4]. The results indicate that none of the evaluated patches yields a significant speedup; all observed performance differences are within 2%, a range where measurement noise can have a substantial impact. We speculate that this limited impact is likely due to the maturity of LLVM. Having been developed and refined for decades, LLVM already incorporates a highly optimized compilation pipeline, making it extremely challenging to achieve substantial performance improvements through a small number of peephole optimizations alone. To further investigate this hypothesis, we compared a recent LLVM version with one released approximately one year earlier; the results are also shown in Figure 5. This comparison similarly reveals no significant runtime performance difference, lending additional support to our speculation.

5.2 The Key Insight and Future Work

The key insight of this work is that, given a reliable and efficient mechanism for rigorously verifying the correctness of optimizations, we can leverage the creativity of LLMs to discover or even directly perform optimizations without being hindered by their inherent unreliability. In this work, we demonstrate the feasibility of this idea by using LLMs to detect missed peephole optimizations in LLVM IR, despite its maturity. Moreover, the fact that LPO is able to identify 62 missed peephole optimizations suggests that state-of-the-art LLMs already possess sufficient code understanding and reasoning capabilities to contribute meaningfully to compiler optimization tasks.

Several promising directions remain for future work. First, although this work focuses on LLVM IR, the underlying idea also applies to other IR (e.g., MLIR) and instruction set architectures (e.g., x86, ARM, and RISC-V). Adapting LPO to these additional targets is therefore an important next step. Second, LPO currently targets only peephole optimizations; future studies could investigate whether LLMs can also help identify missed cases in other classes of optimizations. Third, while LPO is designed to search for missed peephole optimizations, the subsequent steps—generalizing optimization patterns and implementing them in the compiler—are still performed manually by maintainers. Future research could explore how LLMs might also assist with these generalization and implementation phases. Finally, rigorous verification tools are essential to the success of LPO, as they provide strong correctness guarantees for LLM-generated outputs. Accordingly, future work may also focus on developing translation validation tools for additional targets or enhancing existing tools (e.g., Alive2).

[4]We did not aggregate all patches into a single comparison, as they were committed over time. Cherry-picking them onto a single LLVM version is infeasible due to frequent changes in the related codebase.

6 Related Work

In this section, we discuss the studies related to this work.

Superoptimization. A superoptimizer takes a sequence of instructions as input and searches for a more efficient, semantically equivalent or refined program. Massalin [26] first introduced the concept of superoptimization and developed the earliest superoptimizer based on exhaustive enumeration, which, while pioneering, was both inefficient and unsound. To address scalability limitations, subsequent work proposed alternative search strategies, often leveraging SMT solvers for equivalence verification. Joshi et al. developed Denali [18], a goal-directed superoptimizer that employs expert knowledge to encode equality-preserving transformations, thereby constraining the search space. Bansal and Aiken [7] presented a superoptimizer that enumerates only canonical instruction sequences to reduce redundancy and stores computed optimizations in a database to avoid repeated work. Schkufza et al. introduced STOKE [35], a superoptimizer for x86-64 binaries that formulates superoptimization as a stochastic search problem and applies Markov Chain Monte Carlo (MCMC) sampling to efficiently explore the vast program space. This approach sacrifices completeness but significantly increases the diversity of candidate programs, thereby improving the quality of the resulting code. Sasnauskas et al. proposed Souper [34], a superoptimizer for LLVM IR that extracts integer-typed functions from a module, traverses backward along dataflow edges from the return instruction, and synthesizes more performant equivalents. Souper is designed to be sound, but its applicability is limited to a purely functional, control-flow-free subset of LLVM IR. Liu et al. proposed Minotaur [21], another superoptimizer for LLVM IR that focuses on optimizations involving integer and floating-point SIMD code. Minotaur supports more operations compared to Souper, but its effectiveness is still constrained by the synthesis-based search strategy. Hydra [30] addresses the limitations of ungeneralized peephole optimizations produced by superoptimizers by generalizing them using program synthesis.

Comparison with LPO: Superoptimizers are capable of discovering new optimization patterns, but they are computationally expensive, thus often limited to tiny window size or specific instruction sets. They may also fail to produce results within reasonable time. In contrast, LPO leverages LLMs to explore optimization opportunities, which is less restricted by the code size and instruction sets, and thus can potentially detect more diverse missed optimizations.

Differential Testing for Missed Optimizations. Differential testing [28] compares the outputs of two or more programs on identical inputs to detect discrepancies, which

may reveal bugs. Barany [9] applied this technique to compilers, identifying missed optimizations by comparing code generated by different compilers for the same source program. Theodoridis et al. [36] extended this approach to dead code elimination, while Liu et al. [22] used differential testing to evaluate WebAssembly optimizers by comparing their outputs with those of modern C compilers. Italiano and Cummins [17] further advanced differential testing for code size optimizations, proposing methods such as comparing outputs from seed and mutated code, evaluating different optimization settings, and analyzing outputs from different compiler versions or distinct compilers.

Comparison with LPO: Differential testing is effective for discovering missed optimizations, but it is limited to optimizations that have been implemented in other compilers or that are not generalized enough. In contrast, LPO can potentially discover new optimizations that have not been implemented in any compiler.

LLM-Based Approaches for Optimization. Recent advances in LLMs have demonstrated their promise in program synthesis, including code generation and optimization tasks. A number of studies have explored the use of these models for optimizations [11, 14–17]. Some of them focus on source-level code optimization and do not address the problem of finding missed optimizations [14, 15]. Cummins et al. [11] investigated the use of LLMs to generate optimized LLVM IR and recommend compiler passes, highlighting their potential but also their limitations in terms of reusability and computational cost. Grubisic et al. [16] incorporated feedback from optimizer-generated code to enhance optimization quality.

Comparison with LPO: The only prior work that aims to detect missed optimizations with LLMs is by Italiano and Cummins [17]. Their approach employs LLMs to mutate code for differential testing, achieving effective results but restricting the discovery of novel optimization patterns. In contrast, LPO leverages the code understanding and reasoning capabilities of LLMs to directly detect suboptimal code snippets, which can potentially discover new unimplemented optimization patterns.

7 Conclusion

This paper introduced LPO, a novel framework that bridges the creative capabilities of LLM with the rigorous guarantees of formal verification to tackle the difficult challenge of discovering new peephole optimizations. Our methodology overcomes the limitations of previous approaches by integrating an LLM-based optimizer within a closed-loop system, where verification feedback iteratively refines the LLM's search for correct and efficient code transformations. Through a comprehensive evaluation on the LLVM compiler, we demonstrated LPO's effectiveness in finding a substantial number of previously unreported optimizations. The fact that many of these have been confirmed and fixed in the official LLVM codebase highlights our framework's practical significance and its ability to outperform state-of-the-art superoptimizers.

Looking ahead, this research opens up exciting new avenues for the use of LLMs in compiler design and beyond. The success of LPO suggests that similar hybrid methodologies, which combine the generative power of LLMs with rigorous formal methods, could be applied to other complex, pattern-based optimization problems. As the reasoning abilities of LLMs continue to advance, we believe that frameworks like LPO will become an indispensable tool for continuously improving the performance of modern compilers, paving the way for more efficient and robust software.

Acknowledgments

We thank all the anonymous reviewers of ASPLOS'26 for their helpful feedback and constructive comments. We also thank the LLVM maintainers and contributors, especially Yingwei Zheng, for their efforts in addressing the reported missed optimizations and for their valuable feedback. This research is partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) through the Discovery Grant, CFI-JELF Project #40736, and an unrestricted gift from Google.

References

[1] [n. d.]. InstCombine contributor guide — LLVM 22.0.0git documentation — llvm.org. https://llvm.org/docs/InstCombineContributorGuide.html. Accessed 01-08-2025.

[2] [n. d.]. LLVM InstCombine Pull Requests on GitHub. https://github.com/llvm/llvm-project/pulls?q=is%3Aopen+is%3Apr+label%3Allvm%3Ainstcombine. Accessed 01-08-2025.

[3] [n. d.]. LLVM Language Reference Manual — LLVM 22.0.0git documentation — llvm.org. https://llvm.org/docs/LangRef.html. Accessed 01-08-2025.

[4] [n. d.]. Overview - CPU 2017 — spec.org. https://www.spec.org/cpu2017/Docs/overview.html. Accessed 12-01-2026.

[5] [n. d.]. The LLVM Compiler Infrastructure Project — llvm.org. https://llvm.org. Accessed 01-08-2025.

[6] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison Wesley.

[7] Sorav Bansal and Alex Aiken. 2006. Automatic generation of peephole superoptimizers. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems.* ACM, San Jose California USA, 394–403. doi:10.1145/1168857.1168906

[8] Sorav Bansal and Alex Aiken. 2008. Binary Translation Using Peephole Superoptimizers. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, Richard Draves and Robbert van Renesse (Eds.). USENIX Association, 177–192. http://www.usenix.org/events/osdi08/tech/full_papers/bansal/bansal.pdf

[9] Gergö Barany. 2018. Finding missed compiler optimizations by differential testing. In *Proceedings of the 27th international conference on compiler construction.* 82–92.

[10] Sebastian Buchwald. 2015. *Optgen: A Generator for Local Optimizations.* Lecture Notes in Computer Science, Vol. 9031. Springer Berlin Heidelberg, Berlin, Heidelberg, 171–189. doi:10.1007/978-3-662-46663-6_9

[11] Chris Cummins, Volker Seeker, Dejan Grubisic, Mostafa Elhoushi, Youwei Liang, Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Kim Hazelwood, Gabriel Synnaeve, and Hugh Leather. 2023. Large Language Models for Compiler Optimization. arXiv:2309.07062 (Sept. 2023). doi:10.48550/arXiv.2309.07062 arXiv:2309.07062 [cs].

[12] Jack W. Davidson and Christopher W. Fraser. 1984. Automatic generation of peephole optimizations. In *Proceedings of the 1984 SIGPLAN symposium on Compiler construction - SIGPLAN '84*. ACM Press, Montreal, Canada, 111–116. doi:10.1145/502874.502885

[13] Charles N Fischer and Richard J LeBlanc Jr. 1991. *Crafting a Compiler with C*. Benjamin-Cummings Publishing Co., Inc.

[14] Shuzheng Gao, Cuiyun Gao, Wenchao Gu, and Michael Lyu. 2024. Search-based llms for code optimization. *arXiv preprint arXiv:2408.12159* (2024).

[15] Spandan Garg, Roshanak Zilouchian Moghaddam, and Neel Sundaresan. 2023. Rapgen: An approach for fixing code inefficiencies in zero-shot. *arXiv preprint arXiv:2306.17077* (2023).

[16] Dejan Grubisic, Chris Cummins, Volker Seeker, and Hugh Leather. 2024. Compiler generated feedback for large language models. *arXiv preprint arXiv:2403.14714* (2024).

[17] Davide Italiano and Chris Cummins. 2024. Finding Missed Code Size Optimizations in Compilers using LLMs. *arXiv preprint arXiv:2501.00655* (2024).

[18] Rajeev Joshi, Greg Nelson, and Keith Randall. 2002. Denali: a goal-directed superoptimizer. *ACM SIGPLAN Notices* 37, 5 (May 2002), 304–314. doi:10.1145/543552.512566

[19] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE, 75–86.

[20] Chris Arthur Lattner. 2002. *LLVM: An infrastructure for multi-stage optimization*. Ph. D. Dissertation. University of Illinois at Urbana-Champaign.

[21] Zhengyang Liu, Stefan Mada, and John Regehr. 2024. Minotaur: A SIMD-oriented synthesizing superoptimizer. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2 (2024), 1561–1585.

[22] Zhibo Liu, Dongwei Xiao, Zongjie Li, Shuai Wang, and Wei Meng. 2023. Exploring missed optimizations in webassembly optimizers. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 436–448.

[23] Nuno P Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: bounded translation validation for LLVM. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 65–79.

[24] Nuno P Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. 2015. Provably correct peephole optimizations with alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 22–32.

[25] José Antonio Hernández López, Boqi Chen, Mootez Saad, Tushar Sharma, and Dániel Varró. 2025. On Inter-Dataset Code Duplication and Data Leakage in Large Language Models. *IEEE Trans. Software Eng.* 51, 1 (2025), 192–205. doi:10.1109/TSE.2024.3504286

[26] Henry Massalin. 1987. Superoptimizer: a look at the smallest program. *ACM SIGARCH Computer Architecture News* 15, 5 (Nov. 1987), 122–126. doi:10.1145/36177.36194

[27] W. M. McKeeman. 1965. Peephole optimization. *Commun. ACM* 8, 7 (1965), 443–444. doi:10.1145/364995.365000

[28] William M McKeeman. 1998. Differential testing for software. *Digital Technical Journal* 10, 1 (1998), 100–107.

[29] David Menendez and Santosh Nagarakatte. 2016. Termination-checking for LLVM peephole optimizations. In *Proceedings of the 38th International Conference on Software Engineering*. 191–202.

[30] Manasij Mukherjee and John Regehr. 2024. Hydra: Generalizing Peephole Optimizations with Program Synthesis. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1 (April 2024), 725–753. doi:10.1145/3649837

[31] George C Necula. 2000. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*. 83–94.

[32] Amir Pnueli, Michael Siegel, and Eli Singerman. 1998. Translation validation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 151–166.

[33] Nikita Popov. 2025. llvm-compile-time-tracker: LLVM compile-time performance tracking infrastructure. https://github.com/nikic/llvm-compile-time-tracker

[34] Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Gratian Lup, Jubi Taneja, and John Regehr. 2017. Souper: A synthesizing superoptimizer. *arXiv preprint arXiv:1711.04422* (2017).

[35] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic superoptimization. *ACM SIGARCH Computer Architecture News* 41, 1 (2013), 305–316.

[36] Theodoros Theodoridis, Manuel Rigger, and Zhendong Su. 2022. Finding missed optimizations through the lens of dead code elimination. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 697–709.

[37] Yingwei Zheng. 2023. LLVM Opt Benchmark. https://github.com/dtcxzyw/llvm-opt-benchmark. Accessed: 30-12-2025.

[38] Xin Zhou, Martin Weyssow, Ratnadira Widyasari, Ting Zhang, Junda He, Yunbo Lyu, Jianming Chang, Beiqi Zhang, Dan Huang, and David Lo. 2025. LessLeak-Bench: A First Investigation of Data Leakage in LLMs Across 83 Software Engineering Benchmarks. *CoRR* abs/2502.06215 (2025). arXiv:2502.06215 doi:10.48550/ARXIV.2502.06215