Finding Missed Code Size Optimizations in Compilers using Large Language Models

Davide Italiano
Meta Platforms, Inc
Menlo Park, USA
davidino@meta.com

Chris Cummins
AI at Meta
Menlo Park, USA
cummins@meta.com

Abstract

Compilers are complex, and significant effort has been expended on testing them. Techniques such as random program generation and differential testing have proved highly effective and have uncovered thousands of bugs in production compilers. The majority of effort has been expended on validating that a compiler produces *correct* code for a given input, while less attention has been paid to ensuring that the compiler produces *performant* code.

In this work we adapt differential testing to the task of identifying missed optimization opportunities in compilers. We develop a novel testing approach which combines large language models (LLMs) with a series of differential testing strategies and use them to find missing code size optimizations in C / C++ compilers.

The advantage of our approach is its simplicity. We offload the complex task of generating random code to an off-the-shelf LLM, and use heuristics and analyses to identify anomalous compiler behavior. Our approach requires fewer than 150 lines of code to implement. This simplicity makes it extensible. By simply changing the target compiler and initial LLM prompt we port the approach from C / C++ to Rust and Swift, finding bugs in both. To date we have reported 24 confirmed bugs in production compilers, and conclude that LLM-assisted testing is a promising avenue for detecting optimization bugs in real world compilers.

CCS Concepts: • **Computing methodologies** → **Artificial intelligence**.

Keywords: compiler testing, large language models

ACM Reference Format:
Davide Italiano and Chris Cummins. 2025. Finding Missed Code Size Optimizations in Compilers using Large Language Models. In *Proceedings of the 34th ACM SIGPLAN International Conference on Compiler Construction (CC '25), March 1–2, 2025, Las Vegas, NV, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3708493.3712686

CC '25, Las Vegas, NV, USA

ACM ISBN 979-8-4007-1407-8/25/03
https://doi.org/10.1145/3708493.3712686

1 Introduction

Significant effort has been put into testing and fuzzing compilers [1, 20]. A common way to test compilers is to generate random programs that get fed into a compiler, using techniques such as differential testing to validate correctness of the generated binaries [18]. Generating random programs is challenging because of the complex nature of code. Random program generators must be written for each language and maintained as the language evolves with new features. This, in itself, requires a complex piece of software which requires extensive compiler and programming language expertise to develop. For example, CSmith [23], a highly effective random program generator which has been used to identify hundreds of bugs in C compilers, comprises well over 40,000 lines of handwritten code, and requires constantly updating as the language evolves. Such an approach can prove prohibitively expensive for new programming languages, and can limit the effectiveness of testing even popular languages.

Additionally, test cases generated by random program generators are often large and hard to interpret, requiring an additional program reduction stage to make them useful in reporting bugs to compiler developers. This requires further compute and language-specific tooling to be built, for example, using C-Reduce in the case of C programs [19]. As an alternative to generating new programs, mutation testing takes as input a *seed* code and modifies it, such as by mutating the parts of the code which are not executed over a given set of inputs [10]. As with random program generation, developing such tools requires a deep understanding of the target programming language features and static and dynamic analyses.

Prior to the advent of LLMs, there was a strand of research that attempted to leverage machine learning for test case generation [2, 6, 15]. The idea was to substitute the rule- and grammar-based test case generators with a learned generative model that could be stochastically sampled to produce new code snippets. The main advantage of such an approach is the enormous reduction in human effort required to train a model vs build a random program generator. Early work demonstrated some promise in generating plausible and interpretable test cases using deep recurrent neural networks, but the techniques struggled under the load of models that demonstrated a poor grasp of programming language syntax and semantics. For example, in [2], the Long Short-Term

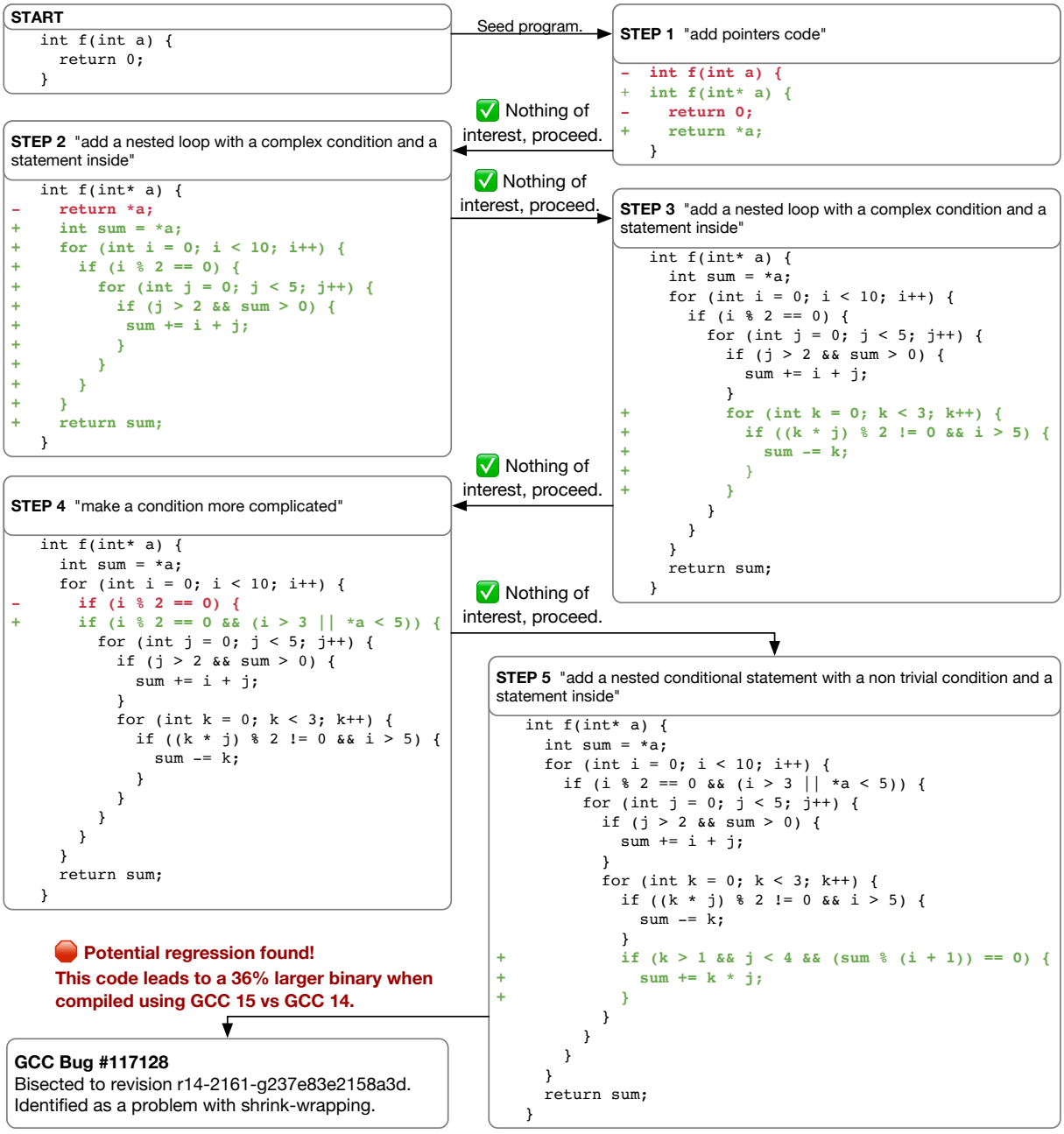

Figure 1. An example of our technique. We instruct an LLM to incrementally mutate a program by randomly sampling a predetermined list of instructions. At each mutation step, an automatic differential testing strategy is used to detect missed optimizations. For this particular example one minute of compute was used and a 36% code size regression was discovered.

Memory network trained on OpenCL required on average 20 attempts to generate a single compilable code snippet.

With LLMs, the capabilities of models to generate and reason about code has improved markedly and there is now a plethora of research directions applying LLMs to different software domains [8]. While nascent, LLMs have already been used to fuzz the correctness of deep learning libraries [3, 4], and C/C++ compilers [22].

While there is a great number of traditional compiler test case generators, and while machine learning and LLMs are being used to simplify their implementation, the automatic generation of compiler test cases to find missed code size

optimizations has received little attention. Yet, code size optimization is critical for embedded computing, mobile applications, and firmware software, and there is active research into novel code optimizations [7, 11, 12]. In this work, we develop novel differential testing methodologies for the express purpose of discovering missed code size optimization opportunities. Our contributions are as follows:

- We present a novel mutation testing methodology which uses large language models to iteratively modify a starting code seed.
- We develop four differential testing strategies for finding missed code size optimizations in compilers.
- We implement our approach in fewer than 150 lines of code and use it to identify 24 bugs in production compilers across C/C++, Rust, and Swift. We release this tool open source.

2 Methodology

In this section we describe how we identify missed code size optimization opportunities in compilers using LLMs.

Figure 1 shows a demonstration of our approach. Starting with a simple seed code, we iteratively instruct an LLM to mutate the code by randomly sampling from a preselected list of mutation instructions. After each mutation, we compile the resulting code and apply a series of differential testing strategies to identify suspicious compilers. Static and dynamic analyses are used to mitigate false positives, and once identified, suspicious compilation results are reported to the user. In this case, after 5 mutations the system identified a 36% code size regression between GCC 15 and GCC 14.

Figure 2 illustrates the workflow of the automated testing methodology. Our approach has two component stages: a method for mutating code, and a series of differential testing strategies to identify potential missed optimization bugs. We describe each in turn, followed by techniques for identifying false positives and detecting duplicate issues.

2.1 Mutating Code Using LLMs

Typical approaches to compiler test case generation requires defining a grammar of the target programming language, and then probabilistically sampling from this grammar, combined with rigorous static and dynamic analyses so as to generate new code which is both syntactically and semantically correct [17, 23]. Such an approach guarantees that generated test cases are free from undefined behavior, but at the expense of complex generation logic. For example, CSmith [23] comprises over 40,000 lines of handwritten code.

We take a different approach. By forfeiting the correct-by-construction guarantee of a grammar-based generator, we are able to use a much simpler engine to generate code for testing. We start with a trivial input program snippet and use an off-the-shelf LLM to rewrite it in such a way as to incrementally add complexity. LLMs make syntactic and semantic errors, and this can make it more challenging to determine if anomalous compiler behavior is indicative of a true bug or a result of a mistake made by in the LLM output. We accommodate for this in two ways: first, because we target missing optimization opportunities, we can permit a greater class of programs than functional tests, and second, by using heuristics and analyses to detect false positives, described in Section 2.3.

Table 1. Instructions used to mutate code. At each step we sample uniformly from this list and generate a prompt which we feed to an LLM.

Control flow
"add a conditional statement with a statement inside"
"add a nested conditional statement with a non trivial condition and a statement inside"
"add a dead conditional statement with a statement inside"
"add a dead nested conditional statement with a non trivial condition and a statement inside"
"add a loop with a complex condition and statement inside"
"add a dead loop with a complex condition and statement inside"
"add a nested loop with a complex condition and a statement inside"
"add a dead nested loop with a complex condition and a statement inside"
Conditionals
"make a condition more complicated"
"make a dead condition more complicated"
Aggregates/pointers
"add array code"
"add pointers code"
"add struct code usage"
"add union code usage"
Function arguments
"add function arguments to a function that already exists, no default arguments"

Seed program. As in prior mutation-based testing approaches, we start with a seed program. Because our iterative approach accumulates mutations, our seed programs can be very simple, shown in Listing 1.

Mutation prompts. At each step of the iterative testing process we build a prompt that instructs the model to mutate the code by randomly sampling from a list of predetermined instructions. We then assemble the instruction and current

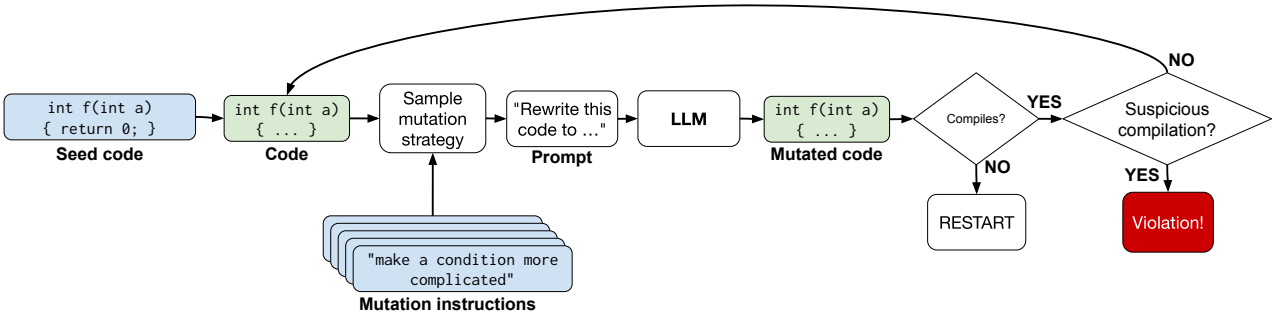

Figure 2. Workflow of the automated testing methodology. The system takes two inputs provided by the user: a seed code and a list of mutation instructions (Section 2.1). Execution iterates until the code mutated by the LLM no longer compiles, or until a series of differential tests and analyses detect a suspicious compilation and trigger a violation (Section 2.2).

```
int f(int a) { return 0; }
```

(a) C / C++

```
#![no_main]
#[no_mangle]
pub fn f(a: i32) -> i32 { 0 }
```

(b) Rust

```
func f(a: Int) -> Int { return a }
```

(c) Swift

Listing 1. Seed programs for different programming languages, used as the starting point for mutation. In all three languages the seed code contains a single empty function with an integer argument. From this, the LLM incrementally expands the scope and complexity of the code, directed by our mutation prompts.

```
Given the following ${language} program,
please ${instruction}:

${code}
```

Listing 2. Template used to generate LLM prompts.

code state into a prompt using the template shown in Listing 2, and feed it to an off-the-shelf LLM, which responds with mutated code.

In this work we use the freely available Llama 3.1 models [5]. These models have been pre-trained and instruction fine tuned on vast corpora of data, with a "knowledge cutoff" of December 2023. For language features added late, one would need a more recently trained LLM.

Mutation instructions. We sample uniformly from a list of predetermined mutation instructions, shown in Table 1. We curated several different kind of instructions, that can be divided into four logical categories:

Control flow modifications: we instruct the model to inject some new control flow into the code. This comes in two form: loops or conditional controls statements (`if` statements). Additionally we control the nesting factor asking the model to generate either a top-level structure or a nested structure inside existing code.

Aggregate mutations: we instruct the model to add code that contains aggregate structures. In our code we take in consideration 4 classes of aggregates: arrays, unions, structures and classes. For languages that don't support C-style unions, e.g. Swift, we replace them with enumerations.

Pointer code: We instruct the model to add code that contains pointers to variables defined in the code.

Mutation of existing conditionals: if `for` loops and `if` statements are already present in the program, we instruct the model to modify the existing condition to make it more complicated without changing the semantics. For example, a transformation that the model has made in response to this prompt is: `if (x == 10){...}` ⟶ `if (x >= 10 && x <= 10){...}`.

Many of the code mutation instructions come in two flavors: *dead code* and *live code*, where dead code is a mutation that changes code that is not executed as part of the regular flow of the program.

The list of mutation instructions is user configurable. We chose these instructions to represent a diverse set of program transformations, but they are not comprehensive, and further could be adapted for domain specific uses if needed.

2.2 Differential Testing Strategies for Discovering Missed Optimizations

In this section we describe four strategies for differential testing that we employ to identify potential missed optimization opportunities. Once a strategy is selected it is kept constant for the duration of an iterative mutation testing session.

2.2.1 Dead Code Differential Testing.

The idea with dead code differential testing is that since our initial seed code evaluates to a constant, the addition of dead code should not change the semantics of the program, so the code generated should be the same. To perform this type of differential test, we disable all mutation instructions except for those which instruct the LLM to insert or modify dead code. The idea is that of telling the LLM to keep adding code and then recompile and compare the code that gets generated. If the code generated changes, it means the compiler failed to prove that the code was dead, pessimizing. This differential testing strategy is particularly interesting because it doesn't need another compiler to be tested against. One can just build the same program twice with the same compiler and compare the code that gets generated.

2.2.2 Optimization Pipeline Differential Testing.

The second differential testing strategy is based on the intuition that certain optimization pipelines try to minimize a metric at the expense of everything else. For example, there is an optimization pipeline that tries to minimize code size at every cost, even at the expense of performance. For LLVM and GCC this is enabled using `-Oz`, and for Rust using `opt-level=z`. We exploit this for differential testing by comparing the code generated by the same compiler using different optimization pipelines (e.g. `-O3`), and triggers a violation if the size of the code at an optimization level different from `-Oz` is smaller from the one at `-Oz`, multiplied by some configurable sensitivity threshold. We found a sensitivity threshold of 5% to be effective.

2.2.3 Single-compiler Differential Testing.

The third differential testing strategy relies on comparing the code generated between different versions of the same compiler. This is useful to find size optimization progressions and regressions. We compare already-released versions of a compiler against the bleeding edge version (trunk/nightly, depending on the language). If the code size regresses, a violation is triggered - a potential candidate for reporting a problem.

2.2.4 Multi-compiler Differential Testing.

The forth strategy is a "basic" differential testing methodology. We compare, when available, multiple different compilers to compare the code that gets generated and if the sizes are sufficiently different, we can flag the one that has larger size as a violation and potential indicator that this compiler misses and optimization found by the other compiler. We use a threshold of 10% to establish "sufficiently different".

2.3 Detecting False Positives

The limitation of using an LLM to write code instead of a grammar-based generator that has full control is that there's the risk of introducing bad examples. While we are unable to fully eliminate the possibility of incorrect code generated by an LLM, we found that we can sufficiently negate the risk using heuristics and validation tools such that we have yet to encounter a false positive that has not been detected by the three techniques described here:

Dead code detection. Of course, when we instruct the LLM to insert dead code, we can't guarantee that the LLM will obey the instruction, and we need a way to verify that the mutated code truly contains only dead code. The problem in general is undecidable, but we can use a proxy metric. Since we control the function signature of the mutated code (e.g. `f()` for C/C++), we can compile an executable program from the test code by inserting a main method that calls it. We run the generated program with counter instrumentation (for coverage) and we wait until the program terminates (if it does), then compare the inserted code and make sure the counter values are identical. If the LLM corrupts the function signature, compilation will fail and the process will restart. There are some limitations to this approach – debug info mapping isn't always reliable, so we might end up mapping from an instruction in assembly to the wrong line, causing false positives and negatives, or the program might never terminate. Nevertheless, we did find these to be effective for our use case, and have yet to encounter a real false positive using this strategy.

Sanitization: Every test case generated is compiled with sanitizers to rule out some classes of potential memory violation problems and undefined behavior. For C/C++, we use UBSan and ASan from LLVM [9]. To reduce the set of false positives further, when available, we run the examples through Compcert [13] that in its "interpreter" mode finds undefined behavior. In our experiments we found that these sanitization steps rejected less than 10% of candidates.

Monotonically increasing size: This is a static heuristic, and it's the easiest one to verify of the all three, so it's the first one we end up employing. It has a higher rate of false negatives. The idea behind it is that in some cases of undefined behavior, the compiler is free to label code as unreachable, and remove. Since our iterative mutation testing incrementally adds complexity for programs with undefined behavior, the optimizer tends to "remove code", rather than adding new code. Therefore, code size can be used as a proxy metric for complexity.

2.4 Detecting Duplicates

As in all undirected test case generators for compilers, our approach may yield numerous unique test cases that trigger violations, but that are duplicates of the same underlying bug. We employ two techniques to help mitigate this.

Release screening: First, we look at the code and we compile with publicly available releases of compilers (for GCC, e.g. 12, 13, 14; for LLVM, e.g. 15.0, 16.0. 17.0). That gives us a first degree of confidence that if the set of versions where the bugs appear overlaps but is not exactly the same, that this suggests two different bugs.

Commit screening/bisection: For regressions, we know that there's a "known working version of the compiler" that doesn't exhibit the behavior. We exploit this to bisect and point to the first commit that introduced the violation. If the commit that introduces the new behavior is different for two different programs, then that very likely points to two different bugs.

3 Finding Bugs in C / C++ Compilers

In this section we evaluate the effectiveness of our iterative mutation testing approach at finding missed optimization opportunities in C / C++ compilers.

3.1 Experimental Setup

To produce the experimental results in this section we ran each of the four differential testing strategies described previously for 8 hours. We use Llama 3.1 [5] as the backing LLM, served on server-grade GPUs. We used two configurations of Llama 3.1: 70B parameters and 405B parameters.

In total we evaluate eight unique configurations of differential testing strategy and model for a total of 64 compute hours of testing. For each configuration we run a single threaded test loop, working through the iterative mutation testing process shown in Figure 2. Code mutation episodes stop after 10 mutations if no violation has been found. The process then repeates until the time is used up.

3.2 Results

Table 2 shows the results of the experiment. We evaluate each configuration using four metrics. The first, *Total programs*, records the number of completed iterative tests. As can be seen, the 70B model produced significantly more programs than the 405B. *Compilable* is the subset of Total programs which did not abort due to a compiler error. We see that the 405B model on average produces fewer compilation errors, but the difference is slight. The average over both models is 96.45% successfully compilable code. *Violations* is the number of times a program triggered the differential testing indicator of a suspicious compile, excluding those filtered out through the false positives detectors. In our experiments we found that the larger 405B parameter LLM is the most effective model to find bugs, demonstrated here by a higher Violations rate. The tradeoff is that inference with the 405B model is more expensive than the smaller 70B model. That means that, for a fixed time period, the 70B model actually produced a higher total number of Violations, simply because the reduced efficiency was offset against far quicker inference, along many more programs to be generated in the same amount of time. Table 3 compares the inference times of the two models. It takes on average 2.92× longer to generate a code mutation using the 405B model than the 70B. During initial development we also experimented with the much smaller 8B parameter Llama model which offers yet faster inference, however, we found that this model so frequently corrupted the code with syntactic or semantic errors that it was impractical, and we abandoned this model configuration.

```
long f() {
  long x = 0;
  while (x < 10) {
    if (x % 2 == 0) { x += 2; }
    else { x += 1; }
  }
  return x;
}
```

(a) Correctly optimized code.

```
long f() {
  long x = 0;
  while (x < 10) {
    if (x % 2 == 0) { x += 2; }
    else { x += 1; }

    if ((x > 20) && (x % 5 == 0)) { x -= 5; }
    if ((x < -5) && (x % 3 == 0)) { x += 3; }
  }
  return x;
}
```

(b) Mutated code.

Listing 3. The addition of the two dead conditionals in (b) exposed a regression in GCC where Value Range Analysis fails to prove that the code is dead.

3.3 Discovered Bugs

During development of our approach we discovered and reported 24 bugs in production compiles. The total compute time for our testing was about one week. In this section we present a representative sample of bugs found.

3.3.1 Bugs Found by Dead Code Differential Testing.

We discovered and reported 5 bugs where compilers failed to identify and remove dead code (Section 2.2.1).

GCC bug 116753. When the code in Listing 3a is compiled using the optimization pipeline `-Os` using GCC trunk, the compiler is able to prove that the loop evaluates to a constant and simplifies the whole computation. But, if two dead conditions are added, as in Listing 3b, the optimizer is not able to optimize this code anymore. This is a regression in value range analysis, as this code used to work with an older version of GCC (12.4.0). The range analysis infrastructure has been reworked in GCC and the new pass can't eliminate the constraints anymore.

LLVM bug 112080. In Listing 4, the `ConstraintElimination` pass in LLVM is not able to find that the inner loop is dead

Table 2. Results from 8 hours of automatic testing on each of the four modes of detecting suspicious compilations. The experiment is repeated using two configurations of the Llama 3.1 model: 70B parameters and 405B parameters. Results show that the larger 405B model has a higher rate of generating violations, but in absolute terms the slower inference means that for a given time budget, the smaller 70B model will discover a greater number of violations.

	Model	Total programs	Compilable	Violations	Avg. steps (min / max)
Dead code differential testing	70B	5,598	5,309 (94.84%)	131 (2.34%)	5.03 (2 / 10)
	405B	1,683	1,598 (94.95%)	40 (2.38%)	4.17 (2 / 7)
Optimization pipeline differential testing	70B	6,346	5,905 (93.05%)	145 (2.28%)	4.89 (1 / 10)
	405B	1,933	1,769 (91.52%)	59 (3.05%)	3.44 (2 / 7)
Single-compiler differential testing	70B	6,305	5,966 (94.62%)	295 (4.68%)	3.57 (1 / 10)
	405B	1,874	1,760 (93.92%)	124 (6.62%)	2.67 (1 / 10)
Multi-compiler differential testing	70B	10,008	9,816 (98.08%)	3,080 (30.78%)	2.48 (1 / 10)
	405B	2,889	2,786 (96.43%)	1,035 (35.83%)	2.30 (1 / 10)

Table 3. The minimum, mean, and max inference times of the two Llama 3.1 model configurations. Measurements aggregated from 32 hours of testing for both models.

Model	Min	Mean	Max
70B	0.37s	4.18s	44.6s
405B	2.34s	12.23s	54.8s

```
long f() {
    long x = 0;
    while (x < 10) {
        while ((x > 20) && (x % 5 == 0)) {
            x -= 5;
        }
        if (x % 2 == 0) {
            x += 2;
        } else {
            x += 1;
        }
    }
    return x;
}
```

Listing 4. The `ConstraintElimination` pass in LLVM is not able to find that the inner loop is dead and remove it. After reporting, this has been fixed upstream.

and remove it. After we reported the LLVM developer provided a patch that fixed the problem, and run this on the testsuite, showing positive results on real-world benchmarks.

3.3.2 Bugs Found by Optimization Pipeline Differential Testing. We reported 4 bugs where optimization pipelines that are intended to reduce code size produce larger binaries than pipelines targeting runtime performance (Section 2.2.2).

```
void f() {
    int arr[5] = {1, 2, 3, 4, 5};
    for (int k = 0; k < 5; k++) {
        arr[k] *= 2;
    }
}
```

Listing 5. This code is optimized to a constant by LLVM when compiled using pipeline `-O3`, but `-Oz` fails to optimize it and generates a loop.

LLVM bug 111571. The code in Listing 5 is fully optimized by LLVM trunk to a constant when the program is compiled with `-O3`, but `-Oz` fails to optimize it. This is because `-O3` runs a loop pass that fully unrolls the loop, and later on, the dead store elimination pass finds out that all the assignments to the array are dead, removing this code. `-Oz` doesn't run the unrolling pass and it's not able to prove this fact, leaving the code not optimized and blowing up the size.

GCC bug 117033. The code in Listing 6 shows that GCC trunk generates larger code at `-Oz` than it generates at `-Os`. In particular, GCC at `-O3` can fold this code to a constant, while `-Os` generates a loop. The `-Os` pipeline is inconsistent about copying the loop header of the inner loop. That copy is critical to remove the outer loop, and subsequently, the Sparse Conditional Constant Propagation pass can't figure out the value using the dominator tree for the function.

3.3.3 Bugs Found by Single-compiler Differential Testing. We reported 12 code size regression bugs found by differential testing a compiler against older revisions of itself (Section 2.2.3).

GCC bug 117123. The code in Listing 7 shows that GCC trunk generates larger code at `-Os` than it does on GCC 13.3. The bisection points to *scccopy*, a new optimization pass for copy propagation and PHI nodes elimination, developed in

```
long f() {
    long x = 0;
    for (int i = 0; i < 5; ++i) {
        while (x < 37) {
            if (x % 4 == 0) {
                x += 4;
            }
        }
    }
    return x;
}
```

Listing 6. This code is optimized to a constant by GCC's `-O3` pipeline but not by the `-Os` pipeline.

```
int f(int a) {
  int arr[5];
  for (int i = 0; i < 5; i++) {
    if (i % 2 == 0 && a > 5 || i % 3 == 0 && a <
     -2) {
      arr[i] = a + i * 2;
    } else {
      arr[i] = a + i;
    }
  }
  return arr[2];
}
```

Listing 7. This code exposed a bug in the partial redundancy elimination pass of GCC, causing a code size regression between versions 13 and 14.

GCC 14. Disabling this pass generates the same code on both versions of the compiler. The GCC developers investigated this bug finding out that *scccopy* just exposes a deficiency in the later partial redundancy elimination (PRE) pass, that misses an equivalence between PHI notes while doing value numbering. The developers fixed this bug on trunk.

GCC bug 117128. The code in Listing 8 discovered a bug where GCC generates larger code at `-Os` on trunk than GCC 14. The change that exposes this fact is a modification in the loop invariant code motion pass, but it's not the real culprit for the regression. In fact, trunk does some register allocation shrink-wrapping that doesn't happen on GCC 14. The GCC developers do concur that if shrink-wrapping in this case regresses size, it shouldn't be done at `-Os`.

3.3.4 Bugs Found by Multi-compiler Differential Testing.

We reported 3 missed optimization bugs found by comparing the binary sizes of code compiled using different compilers (Section 2.2.4).

GCC bug 116868. The code in Listing 9 bug shows that GCC can't prove that this function returns a vector with a

```
struct Potato {
  bool isMashed;
};
void dont_be_here();
int patatino_a;
void patatino() {
  if (patatino_a && patatino_a % 2 == 0 &&
   patatino_a != 10)
    ;
  else {
    Potato spud;
    int spud_0 = patatino_a;
    spud.isMashed = false;
    for (int k = 0; patatino_a == 0 && spud_0 >
     1000; k++)
      for (int l = 0; l < 5; l++)
        spud_0 += l * k;
    for (; spud_0;)
      dont_be_here();
  }
}
```

Listing 8. This code uncovered a regression in GCC 14 caused by the interaction of loop invariant code motion and shrink-wrapping passes.

```
#include <vector>
int sumVector() {
    const std::vector<int> vec = {1};
    int sum = 0;
    for (int i = 0; i < vec.size(); i++) {
        sum += vec[i];
    }
    return sum;
}
```

Listing 9. Clang will optimize this code to a constant. GCC lacks the required analysis to perform this optimization. The GCC developers are adding this feature to reach parity.

single element, that's constant. This is because GCC can't safely prove that the allocation (via new) is "sane" (as defined by the C++ standard). Clang performs the expected optimization because it defaults to `-fassume-sane-operator-new` as frontend flag, which asserts that a call to the operator `new` has no side-effects beside the allocation, in particular that doesn't inspect or modify global memory. GCC just fixed this bug implementing the support. CSmith wouldn't be able to find this bug as it doesn't generate C++ code which contains calls to the standard library.

3.4 LLM Code Mutation Failure Cases

On average 3.55% of iterative test mutation sessions end because the LLM generated code that does not compile. In these

```
int f(void) {
  int a = 0;
  for (;;) {
    a += 1;
    a -= 1;
  }
}
```

Listing 10. Incorrect LLM-mutated code in respond to a prompt instructing it to produce a dead loop. Here the loop has no side effects, but is not dead code since it would be reached during the normal flow of execution.

cases we simply revert to the original seed code and restart mutation testing. Sometimes, the LLM will omit correctly compilable code but with unwanted or invalid semantics. The most common of these errors we observed is failure of the model to interpret the meaning of "dead code". We notice that sometimes the model interprets dead code as "code that does nothing". An example of this failure is shown in Listing 10. In these cases, we use the dynamic instruction counts to filter out these invalid test cases, as described in Section 2.3.

4 Extending to Other Languages

Because our technique uses an off-the-shelf LLM and automatic validation tools, it can easily be adapted to other languages by simply changing the prompt and the target compiler. To provide a preliminary evaluation of the extensibility of this approach we ported our 150 line Python script for testing C / C++ compilers to Rust and Swift. Adapting to each language is a trivial code change. We need only change the compiler invocation, and change the initial seed program as described in Section 2.1. We chose to target Rust and Swift languages as they are relatively young and so do not have the same infrastructure for random test case generation.

4.1 Experimental Setup

We modified the system to support Rust and Swift in turn by changing the language in the LLM prompt, and the compiler invocation commands. We then ran a single threaded instance of the testing loop for one hour each.

4.2 Results

Within a single hour of automated testing we discovered one bug in Swift and four bugs in Rust.

Rust bug 130421. The code in Listing 11 produces a fully vectorized loop for something that can be simplified. This happens because the compiler relies on the loop to be fully unrolled to find out that the sum of the values is constant. Instead, the loop is just partially unrolled and vectorized,

```
pub fn foo() -> i64 {
    let mut result = 0;
    for i in 0..10 {
        for j in 0..10 {
            for k in 0..10 {
                if (i + j + k == 10) {
                    result += 1;
                }
            }
        }
    }
    return result
}
```

Listing 11. The Rust compiler fails to fully simplify these nested loops because it relies on the loop to be fully unrolled to find out that the sum of the values is a constant.

```
func f() -> Int {
  var numbers = [1,2,3,4,5]
  var sum = 0
  for number in numbers {
    if (number > 2 && number < 5) ||
       (number == 1 || number == 3) ||
       (number % 2 == 0 && number > 1) ||
       (number % 3 == 0 && number < 4) ||
       (number * 2 == 6 && number + 1 == 4) ||
       (number - 1 == 2 && number / 2 == 1) {
      sum += number
      while (number + 1 == 5 && number * 2 == 9) {
      }
    }
  }
  return sum
}
```

Listing 12. The Swift compiler fails to establish that the nested while loop is dead and will never be reached.

leading to larger code, and also slower computation of the result.

Swift bug 76535. The code in Listing 12 uncovers a bug shows in Swift's compiler using `-Osize` and `-O` optimization pipelines. The compiler fails to establish that the nested while loop is dead and will never be reached. Removing the nested while loop does result in the loop being simplified. This could be caught in Swift if one of the SIL passes (the Swift intermediate language) would implement a range analysis mechanism to remove the dead computations.

Rust bug 132888. The code in Listing 13 when compiled with the Rust nightly compiler is larger than when it is compiled with Rust 1.81.0. The backend of the compiler decides

```
#![no_main]
#[no_mangle]
pub struct Point {
    x: i32,
    y: i32,
}

#[no_mangle]
pub fn f(a: Point) -> i32 {
    if a.x > 0 && a.y < 0 || a.x < 0 && a.y > 0 {
        a.x * a.y
    } else {
        a.x + a.y
    }
}
```

Listing 13. The size of this Rust code regressed by 50% between 1.81.0 and trunk.

```
#![no_main]
#[no_mangle]
pub fn f(a: i32) -> i32 { a + a }

#[no_mangle]
pub fn g(a: [i32; 5]) -> i32 {
    let mut sum = 0;
    let arr = [1, 2, 3, 4, 5];
    for i in a.iter().chain(arr.iter()) {
        sum += i;
    }
    sum
}
```

Listing 14. The Rust compiler emits a suboptimal sequence of instructions in the backend for this example.

to emit a different sequence of instructions that increase codesize overall for both `x86-64` and `aarch64`.

Rust bug 132890. The code in Listing 14 shows a simple array iteration, and results in a larger binary when compiled with Rust nightly than when compiled with Rust 1.73.0. This is another example of the compiler backend lowering a suboptimal sequence of instructions. The model in this example the LLM generates code with non trivial API usage (e.g. `chain()`, `iter()`).

5 Related Work

Automatic test case generation for compilers is a well established technique for compiler validation and has been surveyed in [1, 20]. Typically random program generators are in themselves complex pieces of code. For example, CSmith [23] is over 40,000 lines of handwritten C++. We take a different approach. Inspired by mutation-based approaches such as equivalence modulo inputs testing [10], we adopt a process of mutation testing, but starting from a trivial seed program, and using an LLM as the engine for rewriting code. Our work differs from previous mutation-based approaches in that it does not require existing libraries to begin mutation, such as the `libc` functions from FreeBSD used in [14].

Prior works using machine learning to synthesize compiler test cases include [2, 6, 15, 22]. Of those, the closest to our work is Fuzz4all [22], a recent publication that employs Large Language Models to generate novel compiler tests. Their approach first ingests documentation and example code and uses it to synthesize new test cases to validate functional correctness of the compiler. Our work differs in that it targets missed optimization opportunities rather than functional correctness, requires no documentation as input, and incrementally builds complexity by mutating a trivial starting seed, rather than synthesizing novel test cases from scratch.

While much attention has been placed on validating the functional correctness of compilers, relatively little attention has been targeted on identifying missed optimizations. Two recent works that attempt to identify missed optimizations are [21] and [16]. In [21], missed optimizations are identified by instrumenting the basic blocks of CSmith-generated code using "dead code markers". In [16], a C program is compiled using both x86 and WebAssembly compilers to identify missed optimizations in WebAssembly. Compared to both these works, our approach is not language specific, requires no instrumentation of programs, and is the first work to use machine learning to generate code rather than handcrafted rules.

6 Conclusions

We describe our experience using Large Language Models to identify missed code size optimization opportunities in compilers. We start with C / C++ and found that by orchestrating an off-the-shelf LLM and existing software validation tools, an effective yet remarkably simple approach could be taken, yielding 24 bugs in production compilers. While our initial results are promising, we are just scratching the surface. In future work we will extend the differential testing methodologies to detect missing runtime performance optimizations, explore prompt engineering approaches to improve the effectiveness of the approach further, and mutate larger seed codes. We hope our initial results to inspire interest in this exciting research direction.

References

[1] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A survey of compiler testing. *CSUR* 53, 1 (2020).

[2] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. 2018. Compiler fuzzing through deep learning. In *ISSTA*.

[3] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large Language Models Are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models. In *ISSTA*.

[4] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. 2023. Testing Deep Learning Libraries via FuzzGPT. *arXiv:2304.02014* (2023).

[5] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The Llama 3 Herd of Models. *arXiv:2407.21783* (2024).

[6] Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn&fuzz: Machine learning for input fuzzing. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 50–59.

[7] Ellis Hoag, Kyungwoo Lee, Julián Mestre, Sergey Pupyrev, and YongKang Zhu. 2024. Reordering Functions in Mobiles Apps for Reduced Size and Faster Start-Up. *TECS* 23, 4 (2024).

[8] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2023. Large Language Models for Software Engineering: A Systematic Literature Review. *arXiv:2308.10620* (2023).

[9] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*.

[10] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. *ACM Sigplan Notices* 49, 6 (2014).

[11] Kyungwoo Lee, Ellis Hoag, and Nikolai Tillmann. 2022. Efficient profile-guided size optimization for native mobile applications. In *CC*.

[12] Kyungwoo Lee, Manman Ren, and Ellis Hoag. 2024. Optimistic and Scalable Global Function Merging. In *LCTES*.

[13] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. 2016. CompCert-a formally verified optimizing compiler. In *ERTS*.

[14] Shaohua Li, Theodoros Theodoridis, and Zhendong Su. 2024. Boosting Compiler Testing by Injecting Real-World Code. In *PLDI*.

[15] Xiao Liu, Xiaoting Li, Rupesh Prajapati, and Dinghao Wu. 2019. Deepfuzz: Automatic generation of syntax valid c programs for fuzz testing. In *AAAI*.

[16] Zhibo Liu, Dongwei Xiao, Zongjie Li, Shuai Wang, and Wei Meng. 2023. Exploring missed optimizations in webassembly optimizers. In *ISSTA*.

[17] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random Testing for C and C++ Compilers with YARPGen. In *OOPSLA*.

[18] William M McKeeman. 1998. Differential Testing for Software. *Digital Technical Journal* 10, 1 (1998).

[19] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *PLDI*.

[20] Yixuan Tang, Zhilei Ren, Weiqiang Kong, and He Jiang. 2020. Compiler testing: a systematic literature analysis. *Frontiers of Computer Science* 14 (2020).

[21] Theodoros Theodoridis, Manuel Rigger, and Zhendong Su. 2022. Finding missed optimizations through the lens of dead code elimination. In *ASPLOS*.

[22] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4all: Universal fuzzing with large language models. In *ICSE*.

[23] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *PLDI*.