Termination-Checking for LLVM Peephole Optimizations

David Menendez Santosh Nagarakatte
Rutgers University, USA
{davemm,santosh.nagarakatte}@cs.rutgers.edu

Abstract

Mainstream compilers contain a large number of peephole optimizations, which perform algebraic simplification of the input program with local rewriting of the code. These optimizations are a persistent source of bugs. Our recent research on Alive, a domain-specific language for expressing peephole optimizations in LLVM, addresses a part of the problem by automatically verifying the correctness of these optimizations and generating C++ code for use with LLVM.

This paper identifies a class of non-termination bugs that arise when a suite of peephole optimizations is executed until a fixed point. An optimization can undo the effect of another optimization in the suite, which results in non-terminating compilation. This paper (1) proposes a methodology to detect non-termination bugs with a suite of peephole optimizations, (2) identifies the necessary condition to ensure termination while composing peephole optimizations, and (3) provides debugging support by generating concrete input programs that cause non-terminating compilation. We have discovered 184 optimization sequences, involving 38 optimizations, that cause non-terminating compilation in LLVM with Alive-generated C++ code.

Keywords Compiler Verification, Peephole Optimization, Alive, Termination

1. Introduction

Compilers translate source programs to multiple target architectures while preserving semantics. Modern compilers are complex because they perform numerous optimizations to obtain the best possible performance on modern architectures. Among them, peephole optimizations perform algebraic simplification with local rewriting of the input code.

Peephole optimizations clean up code resulting from other optimizations and also canonicalize code, which enables other optimizations. The LLVM compiler's main peephole optimization pass, *InstCombine*, contains over a thousand optimizations. Typically, the compiler developer observes a pattern that the compiler fails to optimize, and adds a peephole optimization to handle it. Once the new optimization is added to the compiler, the developer runs the regression test suites to ensure correctness.

Unsurprisingly, peephole optimizations are a common source of compiler bugs [19, 26, 38]. These bugs typically occur in corner cases, especially in the presence of undefined behavior. To address this problem of creating correct peephole optimizations, we have developed Alive, a domain-specific language for specifying peephole optimizations in LLVM [26]. An Alive optimization has the form `source ⇒ target`, with an optional precondition. The optimization checks the input code for a pattern of the form $source$ and replaces it with the $target$. Optimizations expressed in Alive are automatically checked for correctness using a Satisfiability Modulo Theory (SMT) solver. Alive provides concrete counterexamples when verification fails, which enables the compiler developer to fix the error. Further, the Alive framework generates C++ code for use within the LLVM compiler to provide competitive compilation time while ensuring consistency of the specification and the implementation. Alive has already detected bugs in existing InstCombine optimizations [26]. There is active interest to replace the InstCombine pass with Alive-generated C++ code.

This paper describes a new class of *non-termination bugs* with peephole optimizations, discovered while translating InstCombine optimizations in LLVM to Alive. The LLVM compiler runs a suite of InstCombine optimizations multiple times until a fixed point. Figure 1 provides a high level overview of the execution of InstCombine optimizations. We observed that certain InstCombine optimizations can undo the effect of other InstCombine optimizations, which can result in non-terminating compilation (*i.e.*, the compiler hangs). When iteratively applying hundreds of optimizations, it is important to consider how they may interact. A developer adding a new optimization cannot be certain of its interaction with existing optimizations without a global view of LLVM's InstCombine optimizations. Hence, a non-termination bug may be undetected for years until some unfortunate programmer discovers it. Further, they manifest only when appropriate input code is provided for compilation.

This paper proposes a methodology to identify compiler non-termination bugs by extending Alive to provide a global view of InstCombine optimizations in LLVM. Our approach explores sequences of optimizations from the Alive suite to see whether they can have bad interactions that cause non-termination bugs when run until a fixed point. Figure 2 illustrates the methodology for detecting compiler non-termination bugs in InstCombine, which is based on composition of sequences of Alive optimizations. The composition of two optimizations is a new optimization that has the effect of applying the two optimizations to an input program, one after the other. Our methodology enumerates all possible sequences of a given length. For each sequence, we determine if it is feasible to compose the sequence of optimizations into a single optimization that summarizes the sequence. As we are specifically interested in the case where subsequent optimizations are enabled by previous ones, we only consider cases where the source of a subsequent optimization matches at least one instruction created by the target of a previous optimization.

function INSTCOMBINEFUNCTION(F)
  repeat
    Add reachable instructions in F to $worklist$
    Remove unreachable blocks from F
    for all $I \in worklist$ **do**
      if Try opt_1: I matches $source(opt_1)$ **then**
        Add all $i \in target(opt_1))$ to $worklist$
        Replace I with $root(target(opt_1))$
      else if Try opt_2: I matches $source(opt_2)$ **then**
        Add all $i \in target(opt_2))$ to $worklist$
        Replace I with $root(target(opt_2))$
      else if Try opt_3: I matches $source(opt_3)$ **then**
        Add all $i \in target(opt_3))$ to $worklist$
        Replace I with $root(target(opt_3))$
        ...
      end if
    end for
  until no changes made
end function

Figure 1: High-level view of InstCombine where opt_1, opt_2, opt_3, and other optimizations are tried in order. The root of the directed acyclic graph is represented as $root$.

Next, we determine if the composed optimization can result in non-termination. The composition itself may be infeasible with some sequences. Even when the composition is feasible, the precondition of the composed optimization of a sequence may not be satisfiable. Such sequences cannot cause compiler non-termination for any input program. When the optimization composes with itself with a satisfiable precondition, the optimization can be applied infinitely many times when the self-composition of the optimization consumes a source program no larger than original optimization (see Figure 4 for an illustrative example). Hence, the necessary conditions for non-termination are: (1) the precondition of the self-composition is satisfiable, and (2) the length of the source of the self-composition is smaller than or equal to the source of the original optimization.

For example, let $O_1O_2O_3$ be an optimization sequence, where O_1, O_2, and O_3 are individual peephole optimizations in the Alive suite. The composition phase generates a single optimization O_z that summarizes $O_1O_2O_3$. The non-termination checker checks if the precondition of O_zO_z is satisfiable and the number of instructions in the source of O_zO_z is smaller or equal to the number of instructions in the source of O_z. If both the conditions are satisfied, the checker reports that the optimization sequence $O_1O_2O_3$ causes compiler non-termination.

When an optimization sequence can cause non-termination, the tool also generates a concrete input to aid debugging. We have discovered 184 optimization sequences involving 38 optimizations that cause compiler non-termination errors in Alive's suite of InstCombine optimizations. We have demonstrated that these optimization sequences cause the generated C++ code for InstCombine to loop indefinitely.

Contributions. This paper:

- Identifies a new class of compiler non-termination bugs, resulting from the lack of a global view of peephole optimizations in LLVM.
- Proposes a methodology to detect compiler non-termination bugs building on top of Alive, which checks the correctness of each individual InstCombine transformation.
- Identifies non-increasing source in the self-composition of a sequence of optimizations as the necessary condition for non-termination.
- Proposes a technique to generate concrete inputs to demonstrate non-termination errors to aid debugging.

Next, we provide a brief background on Alive because we build our termination checker on top of Alive.

2. Background on Alive

Alive is a language for specifying peephole optimizations for LLVM. The Alive interpreter automatically checks the correctness of the optimization using a SMT solver and generates C++ code that implements the optimization, for use in LLVM. Alive syntax is similar to the LLVM IR because the intended users of Alive (LLVM developers) are already familiar with it. In contrast to the LLVM IR, Alive optimizations are parametric over types and bit widths. Hence, the Alive interpreter checks the correctness of the optimization for all feasible types and bit widths (up to a certain bound). Alive abstracts the various kinds of undefined behavior while the interpreter reasons about them during verification. Figure 3 illustrates the process of verifying and generating C++ code with the Alive.

2.1 InstCombine Optimizations in Alive

Alive optimizations have the form `source` $\Rightarrow$ `target`, with an optional precondition. An Alive optimization replaces the root of a directed acyclic graph (DAG) of instructions in the source with the root of a new directed acyclic graph in the target. Hence, the source DAG and the target DAG must have the same root variable (`%r`). An example Alive optimization is given below.

```
Pre: C2 == ~C1
%w = or %p, C2
%x = xor %w, C1
%y = add %x, 1
%r = add %y, %q
  =>
%a = and %p, C1
%r = sub %q, %a
```

In the optimization above, the DAG rooted at `%r` in the source is replaced with the DAG in the target when the precondition is satisfied (*i.e.*, `C2 == ~C1`, where `C1` and `C2` are symbolic constants). In general, Alive preconditions consist of built-in predicates, equalities, and signed/unsigned inequalities. The predicates in Alive are used to represent the results of LLVM's dataflow analyses.

The instructions in Alive are similar to instructions in the LLVM IR. The variables in Alive other than the root are either input variables or temporary variables generated in the source and target. An Alive optimization can also have symbolic and literal constants in the source, target, and precondition. Constant expressions may occur in the target and precondition and can contain constants, arithmetic and bitwise operators, and common math-based built-in functions. In the example above, `%r` is the root of the DAG, `%p` and `%q` are input variables, `C1` and `C2` are symbolic constants, and `%w`, `%x`, `%y`, and `%a` are temporary variables.

The target may refer to instructions defined in the source, redefine them, or create new instructions. When the target redefines an instruction used in the source, it indicates that the target instruction will replace the corresponding instruction in the source. The root instruction in the source of an Alive optimization will always be replaced in the target.

Alive provides abstraction over types. Hence, a single optimization can apply to a wide range of types constrained by the instructions present in the source and target. For example, binary operators require their arguments and their result to be integers of the same bit width. The compiler writer can optionally provide types in Alive to reduce verification time. Types in Alive are a subset of LLVM's

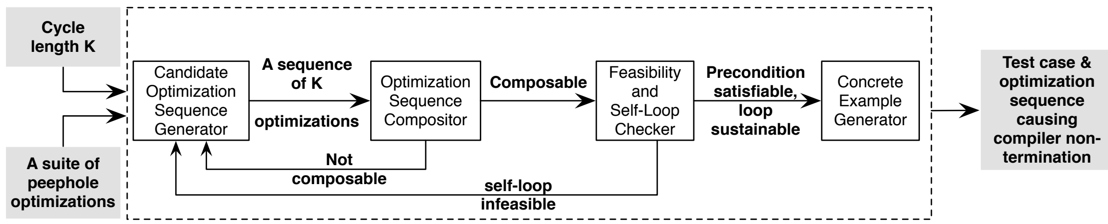

Figure 2: Workflow of our termination checking algorithm.

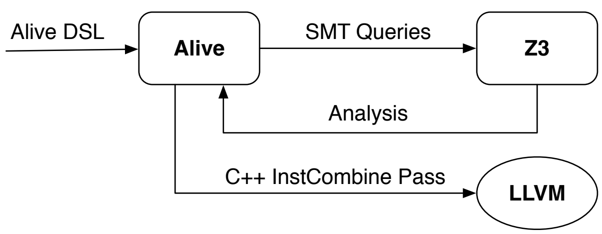

Figure 3: The figure illustrates how the optimization in Alive domain specific language (DSL) is checked for correctness by the Alive interpreter with queries to the Z3 (SMT) solver. On successful verification, the Alive interpreter generates the C++ code for use in LLVM. If the verification is unsuccessful, it generates counter examples with concrete values to illustrate the error.

type system, including integers of various bit widths, pointer types, array types, and void. Alive automatically determines the type constraints implicit in an optimization, and then checks the validity of the optimization for various assignments of types which meet those constraints.

Undefined behavior. Most compiler bugs are a result of misunderstanding semantics, especially regarding various kinds of undefined behavior [26]. Alive's verification engine reasons about the correctness of optimizations in the presence of undefined behavior, which eases the job of the compiler writer. Alive's semantics for instructions is based on the semantics of the LLVM IR. The semantics of an instruction specify when the instruction is well-defined. LLVM optimizes the program with the assumption that the programmer never intends to have undefined behavior in the program.

LLVM instructions have attributes that modify the behavior of the instruction [26]. Examples of such attributes are `nsw` (no signed wrap), `nuw` (no unsigned wrap), and `exact`. An arithmetic instruction with the no signed wrap attribute produces a *poison value* on signed overflows [26]. Poison values produce undefined behavior when such values are used in instructions with side effects. The poison value propagates along dependencies. Hence, any instruction that receives a poison value as input will produce a poison value as output.

2.2 Alive Semantics for Pattern Matching

Statements in Alive have slightly different semantics, depending on whether they occur in the source or in the target. Instructions occurring in the source act as *patterns*, indicating the minimum requirements for a given input to match the source. In particular, the presence of an instruction attribute (*e.g.*, `nsw`) in the source means that the attribute *must* be present in the input for the pattern to match, but a pattern not containing an attribute will match an instruction with the attribute. In contrast, instructions occurring in the target act as *code*. The attributes present in the target are exactly those which will be present in the output of an optimization.

2.3 Correctness and Generating C++ code

Given an optimization, the Alive interpreter uses an SMT solver to help instantiate candidate types for the optimization. The Alive interpreter encodes the Alive optimization with concrete types into first order logic formulae. The validity of the formulae imply the correctness of the optimization. The interpreter generates the following validity checks under the conditions that the source is well-defined and poison-free, and the precondition is satisfied: (1) the target is well-defined, (2) the target is poison-free, and (3) the roots of the source and target compute the same value. These checks are performed for each feasible type instantiation.

When verification succeeds, the Alive interpreter generates C++ code for the optimization using LLVM's PatternMatch support [1]. Automatic generation of C++ code for the optimization provides competitive compilation time and ensures consistency of the specification and the implementation. Next, we will discuss how to detect non-termination bugs in a suite of Alive InstCombine optimizations.

3. Non-termination Detection

Alive verifies the correctness of each individual optimization. Even when all optimizations are individually correct, a suite of them can cause a compiler to experience a non-termination bug. An optimization in the suite can undo the work of other optimizations. When such optimizations are run until a fixed point, the compiler will not terminate. Consider the two optimizations (O_1 and O_2) shown in Figure 4(a) and Figure 4(b), which we will use to illustrate our technique for detecting compiler non-termination. Both optimizations are individually correct, as indicated by the Alive verification engine. However, the compiler will not terminate when the two optimizations are executed until a fixed point.

When an LLVM developer proposes a new optimization for the InstCombine suite, it is necessary to determine whether the newly proposed optimization can interfere with existing optimizations. LLVM InstCombine, which is a collection of C++ code, does not have a global view that could be used to identify non-termination bugs. With Alive being adopted by LLVM developers for InstCombine verification, our strategy is to use the Alive suite to provide a global view of existing peephole optimizations and check if existing optimizations or the newly proposed optimization can cause compiler non-termination.

The optimizations in InstCombine are attempted sequentially one after the other, as shown in Figure 1. Suppose the optimizations in InstCombine are O_1, O_2, O_3. First, the optimization O_1 is attempted. If it is successful, then the newly created instruction is added to the work list and the entire suite of optimizations is tried again, as shown in Figure 1. If it is not possible to apply optimization O_1, then optimization O_2 is attempted as described earlier. If

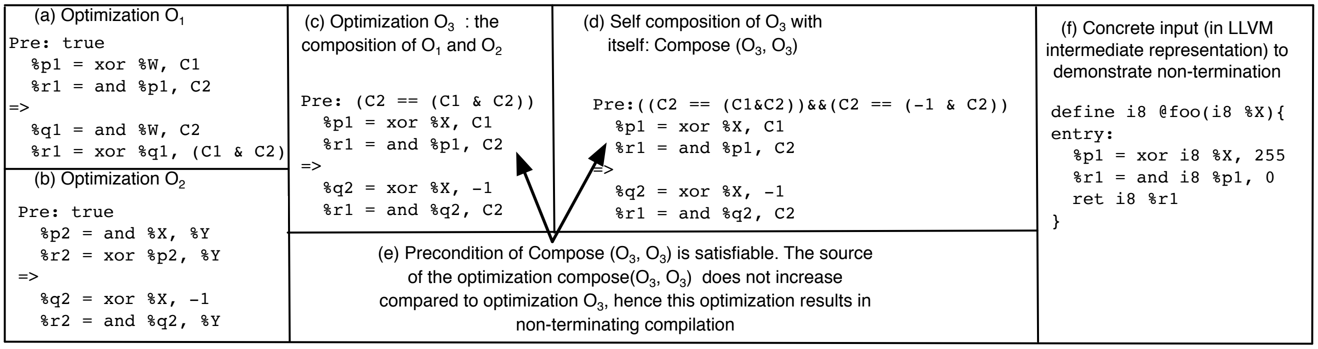

Figure 4: An example illustrating the process of non-termination detection. The optimization sequence generated is $O_1 O_2$ where O_1 and O_2 are optimizations shown in (a) and (b) respectively. Optimization O_3 shown in (c) is the composition of O_1 and O_2. The self-composition (*i.e.*, the composition of O_3 with itself) is presented in (d). The source of the self-composition of O_3 has the same length as the source of O_3, and its precondition is satisfiable. Hence, optimization sequence $O_1 O_2$ can result in compiler non-termination with an appropriate input. The test case generated for debugging is shown in (f).

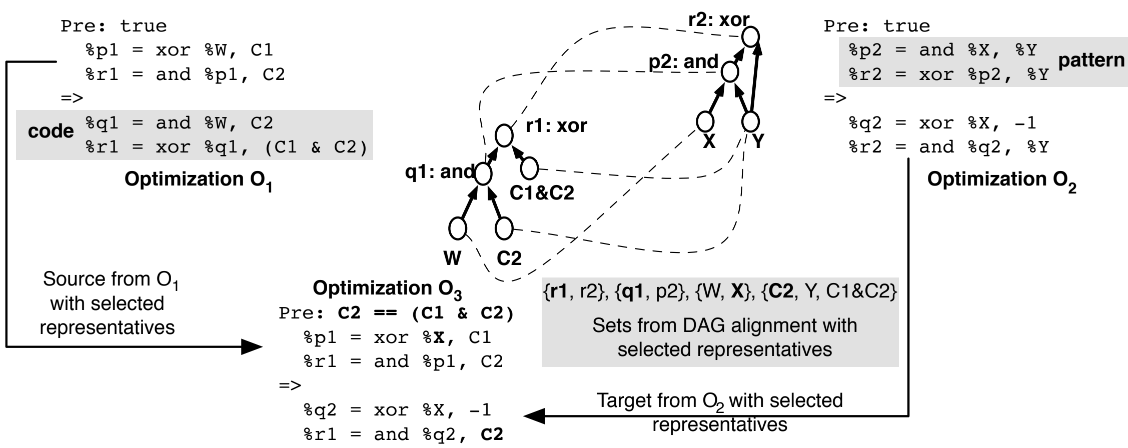

Figure 5: Composing two optimizations O_1 and O_2 to generate O_3. The dashed lines connect the nodes that align with each other in the respective DAG's in the code and the pattern.

optimization O_3 is applicable in a subset of the cases where optimization O_1 is applicable, then O_3 will never be invoked due to this structure of InstCombine optimizations. We say that optimization O_1 *shadows* optimization O_3. In the absence of shadowing, detecting non-termination in InstCombine reduces to the following problem:

Given a suite of InstCombine optimizations where no optimization shadows another, do optimization sequences that cause compiler non-termination exist?

Our general strategy to detect compiler non-termination bugs in the Alive suite consists of three steps. First, we generate sequences of optimizations up to a certain bound ($O_1 O_2$ is the optimization sequence in Figure 4). Second, we compose the optimizations in the sequence to generate a resultant optimization that summarizes the effect of running all the constituent optimizations on the input code, one after the other (see Figure 4(c)). Third, we compose the resultant optimization from the previous step with itself and check if it consumes a larger source pattern and if its precondition is satisfiable (see Figure 4(d) and Figure 4(e)). We describe each of these steps in the following subsections.

3.1 Generating Candidate Sequences

Our goal is to determine whether there exists a sequence of optimizations that can be performed indefinitely. Given a suite of n optimizations, the number of possible optimization sequences with each optimization used exactly once has an upper bound of $O(n!)$. The space of optimization sequences with repetition is even larger. The candidate optimization sequence generation phase explores this large state space in a systematic fashion by iteratively enumerating all sequences up to a certain length (*i.e.*, we explore sequences of length 1, length 2, and so on). We restrict ourselves to sequences where each optimization appears at most once. Each sequence may give rise to zero or more compositions (see Section 3.2). Our approach checks whether the composition of the candidate sequence is feasible. In practice, we have to explore a large number of optimization sequences to find a feasible composition for cycle lengths greater than 6.

```
function COMPOSE(O1, O2)
    Sets ← AlignDAGs(O1, O2)
    CheckValidity(Sets)
    Φ ← ∅
    for all S ∈ Sets do
        φ ← SelectReplacement(S)
        Φ ← Φ ∪ φ
    end for
    src(O3) ← Graft(root(src(O1)), Sets)
    tgt(O3) ← Graft(root(tgt(O2)), Sets)
    pre(O3) ←
            Graft(pre(O1), Sets) ∧ Graft(pre(O2), Sets) ∧ Φ
    return O3
end function
```

Figure 6: Compose optimizations O_1, O_2, if possible.

```
function ALIGNDAGS(O1, O2)
    Sets ← ∅
    worklist ← [⟨root(tgt(O1)), root(src(O2))⟩]
    while worklist not empty do
        ⟨t1, t2⟩ ← pop(worklist)
        for i ∈ {1, 2} do
            if ∃S ∈ Sets such that ti ∈ S then
                Si ← S
            else                              ▷ Create a new set for this value
                Si ← {ti}
                Sets ← Sets ∪ {Si}
            end if
        end for
        if S1 ≠ S2 then
            S3 ← S1 ∪ S2
            if S1 and S2 have code instructions then
                reject
            end if
            if S1 and S2 have pattern instructions then
                ⟨v, pairs⟩ ← Merge(patInstr(S1), patInstr(S2))
                worklist ← append(worklist, pairs)
                patInstr(S3) ← v
            end if
            if S3 has a new pattern-code pair then
                pairs ← Match(patInstr(S3), codeInstr(S3))
                worklist ← append(worklist, pairs)
            end if
            Sets ← (Sets \ {S1, S2}) ∪ {S3}
        end if
    end while
    return Sets
end function
```

Figure 7: Align two DAGs and return the sets of unified nodes. $codeInstr(S)$ and $patInstr(S)$ denote the code instruction and merged pattern instruction for S.

3.2 Composition

Optimization composition is a key step in our procedure for detecting non-termination. When composing two optimizations O_1 and O_2, we attempt to see if O_2 will match the result of applying O_1 to some input. Therefore, we treat the target of O_1 as *code* and the source of O_2 as a *pattern*, as described in Section 2.2. Because the code and the pattern may have multiple instructions, the composition can potentially take place in more than one way. We are interested specifically in cases where O_1 enables O_2, so we require that O_2 match at least one instruction created by O_1.

Figure 6 presents the algorithm for the simple case of root-to-root compositions. Extending the algorithm to handle non-root to root and root to non-root compositions is straightforward; we omit the details due to space considerations. The composition algorithm first determines which values in the source of O_2 (the pattern) must unify with the target of O_1 (the code), which is accomplished by aligning their respective DAGs. DAG alignment results in sets of values that must be identical for the composition to be feasible. The sets generated from DAG alignment are checked to ensure that the sets are valid (*e.g.*, an instruction and a constant cannot be in the same set). Finally, a new optimization is created by expanding the source of O_1 and the target of O_2 with appropriate representatives from the DAG alignment step. Figure 5 illustrates composition of optimizations with DAG alignment.

3.2.1 DAG Alignment

The DAG alignment process finds all values in the pattern and the code that must unify for the composition to occur. We perform DAG alignment using a worklist-based algorithm. The worklist contains the list of values from the respective DAGs that should be processed for unification. The result of the DAG alignment stage is a collection of sets, where each set contains values that must be unified for the composition to occur. The algorithm for DAG alignment maintains the invariant that any unified set contains at most one code instruction, because a pattern variable cannot match more than one distinct code instruction.

Figure 7 provides the algorithm for DAG alignment. Initially, a pair consisting of roots of the respective DAGs in the code and the pattern is added to the worklist. Each value in the respective DAGs will be placed in exactly one set, which will combine with other sets as the DAG alignment proceeds. When an item (a, b) is processed from the worklist, the algorithm retrieves the sets S_a and S_b corresponding to values a and b.

Matching a code and a pattern instruction. If S_a and S_b have one code instruction and one pattern instruction, they are matched with each other. The match algorithm in Figure 8 checks whether the opcodes of the code instruction and the pattern instruction in S_a and S_b are exactly the same. The match algorithm rejects the composition otherwise. The match algorithm rejects the composition if the instruction attributes of the code instruction are not a subset of the instruction attributes of the pattern instruction, according to the Alive pattern matching semantics (see Section 2.2). The match algorithm also adds a tuple for each operand of the matching instructions to the worklist.

Merging pattern instructions. If S_a and S_b both have pattern instructions, they are merged according to the algorithm in Figure 9. The need for merging two pattern instructions arises because two distinct instructions in the pattern may map to the same value in the code (*e.g.* when a code instruction `add %w, %w` is matched with pattern `add %x, %y` where `%x` and `%y` are distinct pattern instructions). The pattern instructions are merged when they perform the same operation and the composition is rejected otherwise. The union of the instruction attributes in the two pattern instructions is computed and used for the merged instruction based on the Alive semantics. The algorithm also adds a tuple for each operand of the merged instructions to the worklist.

Finally, the union of two sets S_a and S_b is added to the list of unified sets and the sets S_a and S_b are removed from the collection of sets, as shown in Figure 7. We use the union-find data structure for the operations on disjoint sets.

Figure 5 illustrates the process of DAG alignment. Initially, a tuple containing the roots (`%r1`, `%r2`) is added to the worklist. When (`%r1`, `%r2`) is processed, the instructions `%r1` and `%r2` are matched, which results in tuples (`%q1`, `%p2`) and (`(C1 & C2)`, `%Y`) being added to the worklist, and a set {`%r1`, `%r2`} is created. When (`%q1`, `%p2`) is processed, instructions `%q1` and `%p2` are

```
function MATCH(v_p, v_c)
    if opcode(v_p) ≠ opcode(v_c) then
        reject
    else if flags(v_p) ⊈ flags(v_c) then
        reject
    else
        return [⟨op_1(v_p), op_1(v_c)⟩, ⟨op_2(v_p), op_2(v_c)⟩, ···]
    end if
end function
```

Figure 8: Match a code instruction against a pattern instruction.

```
function MERGE(v_1, v_2)
    if opcode(v_1) ≠ opcode(v_2) then
        reject
    else
        v_3 ← copy(v_1)
        flags(v_3) ← flags(v_1) ∪ flags(v_2)
        return ⟨v_3, [⟨op_1(v_1), op_1(v_2)⟩, ⟨op_2(v_1), op_2(v_2)⟩, ···]⟩
    end if
end function
```

Figure 9: Combine two patterns, if possible.

matched, which results in (`%W`, `%X`) and (`%C2`, `%Y`) being added to the worklist, and a set {`%q1`, `%p2`} is created. The collection of unified sets generated when all the elements are processed is shown in Figure 5.

3.2.2 Validity of Sets from DAG Alignment

We check the collection of sets obtained from the DAG alignment for well-formedness ($CheckValidity$ call in Figure 6). We check that no instruction in a set has another value in the same set as the operand, either directly or transitively. We perform this check by detecting cycles in the dependency graph constructed between the sets. The nodes of the dependency graph are sets from the DAG alignment stage. Two nodes A and B have an edge if an instruction in set A depends on a value from set B. The composition is not feasible when such circular dependencies arise. We reject such compositions.

We also ensure that no set from DAG alignment has both instructions and constants. Furthermore, no set may contain two distinct specific constant literals (*e.g.*, `1` and `2`). When we are composing O_1 with O_2 with DAG alignment, we also check that no value in the source of O_1 is unified with an intermediate value generated in the target of O_1. These checks ensure that the collection of sets generated from DAG alignment correspond to a possible sequence of Alive optimization applications.

3.2.3 Selection of Replacement for the Sets

Once the collection of sets from the DAG alignment is checked for well-formedness, we select a replacement value for each set. The replacement value will be used in the final stage when creating the composed optimization. The goal of this selection step is to pick the most specific value, according to Alive semantics, for each set. If the set contains roots, then the root of O_1's source is selected. Otherwise, the replacement values are selected in the following priority order: code instruction, merged pattern instructions, specific constants, constant expressions, symbolic constants, and input variables.

Constant expressions (which don't include specific and symbolic constants) cannot occur in the source of an optimization in Alive. When a set contains a constant expression and one of the values in the set occurs in the source of O_1, we choose that value. If the set contains constant expressions not selected as the replacement value, we create equations for the new optimization's precondition.

In Figure 5, the selected replacement value for each set is in bold. The set {`C2`, `%Y`, `(C1 & C2)`} contains a symbolic constant, an input variable, and a constant expression. The symbolic constant `C2` is present in the source of O_1. Hence, the symbolic constant `C2` is chosen as the replacement and we introduce new clauses to the precondition of the composed optimization to enforce the equality of the symbolic constant and the constant expression. In Figure 5, the new equation added is `C2 == (C1 & C2)`.

```
function GRAFT(t, Sets)
    if (∃S ∈ Sets such that t ∈ S) ∧ replacement(S) ≠ t then
        return Graft(replacement(S), Sets)
    else
        t' ← copy(t)
        for all operands i do
            op_i(t') ← Graft(op_i(t), Sets)
        end for
        return t'
    end if
end function
```

Figure 10: Create a new DAG by recursively examining an existing DAG. Values in a unified set are replaced by their representative.

3.2.4 Creating the New Composed Optimization

Finally, the fourth stage creates the new optimization. The $graft$ procedure shown in Figure 10 recursively walks through the dependency graph of its argument, replacing any values from the matching set with the value selected for its set. The $graft$ procedure ensures that newly-created instructions and symbolic constants have unique names. If one or both optimizations includes a precondition, then the graft procedure performs the same substitution on the input precondition(s) to form the new precondition, along with any additional equations produced during selection. Figure 5 presents the final optimization O_3 generated with the graft procedure.

3.3 Necessary Condition for Non-Termination

Our general strategy for identifying non-termination bugs is to compose a given optimization (or a new composed optimization generated from a sequence) with itself. However, the fact that an optimization (or a sequence of optimizations) can self-compose does not necessarily result in non-terminating compilation. Although the self-composition is feasible, such a self-composition may never be invoked if the precondition is not satisfiable.

Moreover, self-composability is necessary, but not sufficient to cause non-termination. Consider the optimization from the Alive suite shown in Figure 11(a), which re-associates addition. The optimization can be self-composed, yielding the optimization shown in Figure 11(b). The precondition of the self-composition is trivially satisfiable. However, the optimization does not result in non-terminating compilation, even though it can be run repeatedly, because the optimization consumes a different fragment of input code each time it runs. Performing the re-associate optimization twice will transform three instructions, instead of two. Performing it three times will transform four instructions, and so on. Thus it can only run a finite number of times on a finite input.

Based on this observation, we consider the size of the source pattern when the optimization is composed with itself to determine if the optimization can result in non-terminating compilation, rather than attempting to determine directly whether an optimiza-

(a)	(b)

`%p = add %a, %b`
`%r = add %p, %c`
`=>`
`%q = add %b, %d`
`%r = add %a, %q`
	`%p1 = add %a1, %b1`
	`%p  = add %p1, %b`
	`%r  = add %p,  %c`
	`=>`
	`%q  = add %b,  %c`
	`%q1 = add %b1, %q`
	`%r  = add %a1, %q1`

Figure 11: (a) Alive optimization that re-associates addition. (b) The self composition of the reassociate add optimization, which does not cause non-terminating compilation because the source pattern of self-composition is larger than the source pattern of the original optimization.

tion decreases code size. Hence, the necessary conditions for non-terminating compilation are:

- The precondition of the self-composition is satisfiable.
- The source pattern of the self-composition is either of the same size or smaller than source pattern of the optimization before the self-composition.

Optimizations O_1 and O_2 in Figure 4 can cause non-terminating compilation because the source pattern of the self-composition in Figure 4(d) is of the same size as the source pattern in Figure 4(c) and the precondition of the self-composition is satisfiable.

4. Debugging Non-termination

The approach described above generates a sequence of optimizations that can cause compiler non-termination. To enable the compiler writer to debug and diagnose the cause of non-terminating compilation, we also generate test cases that demonstrate these non-termination errors. The resultant composition generated from a sequence of optimizations already has sufficient information that can be leveraged to generate a test case, which would enable the compiler developer to debug the error.

As the source and target of an Alive optimization are written in a generalized superset of LLVM IR, specializing the source of the optimization will generate an example test case. Alive provides abstractions over bitwidths and constants in comparison to the LLVM IR. Hence, generating a test case requires identifying a bitwidth for each individual type and generating concrete values for the symbolic constants and constant expressions. Further, the test case must be a self-contained LLVM IR unit (*e.g.*, a function).

The test case for the composed optimization representing the effect of optimizations in a sequence is generated in three steps. First, we specialize the types by choosing an arbitrary type assignment which meets the optimization's typing constraints. The constraints are expressed in first order logic and the resulting formula is provided to an SMT solver. The model obtained from the solver provides the type instantiations for the values in the test case. Second, we specialize the symbolic constants into concrete values respecting the constraints in the optimization's precondition. Similar to types, we express the precondition in first-order logic and query the SMT solver with the resulting formula. The model from the SMT solver provides the concrete constants. Third, we generate a self contained test case in the LLVM intermediate representation. The test case is structured as an LLVM function, with parameters corresponding to the optimization's input variables and return value corresponding to its root. For each instruction in the source, we generate a corresponding LLVM instruction, applying the types and constants obtained in the previous steps.

Figure 4(f) shows the test case generated for the composed optimization in Figure 4(c). The generated test case has `i8` as the

n	Optimization Sequences	Complete Compo-sitions	Self-com-positions	Non-in-creasing	Cycles Found
1	416	416	296	25	23
2	86 320	7 001	4 292	31	27
3	23 824 320	182 678	96 989	49	35
4	7 379 583 120	5 524 634	2 694 291	152	99
5*	13 119 902 905	1 000 000	463 017	2	0
6*	97 613 680 549	1 000 000	394 794	0	0
7*	474 163 216 578	1 000 000	395 638	0	0
Total Number of Cycles					**184**

Table 1: The first and second columns report the length of the cycle in the exploration and the number of optimization sequences that were explored when looking for the n cycle. The third column reports the number of optimizations that result from a complete composition of the sequence. The fourth column reports how many self-compositions of the composed optimizations were possible. The fifth column reports the number of self-compositions that had a non-increasing source. The last column reports the cycles found. A * indicates a randomized search of optimization sequences until a million complete compositions were found.

type chosen for the source variables. The symbolic constants `C1` and `C2` have been instantiated with 255 and 0 respectively because they satisfy the precondition.

Optimizations using results from dataflow analyses. Alive optimizations can use the result of LLVM data flow analyses. Generating test cases that satisfy the results of dataflow analyses is challenging. For example, if the precondition contains a predicate `WillNotOverflowSignedAdd(%a,%b)`, we cannot simply make `%a` and `%b` parameters to a function. LLVM will not be able show that their addition will not result in signed overflow and the optimization will not be applied. To address this challenge, we generate symbolic constants that satisfy the axiomatic specification of the dataflow analyses in Alive. One minor niggle with this approach is that we will have to disable constant folding before running InstCombine in LLVM. For the example above, our test case will contain concrete constants (*e.g.*, 42 and 7, assuming the type of `%a` and `%b` is `i8`) whose addition does not result in signed overflow.

Shadowing of optimizations. An implicit precondition with our generated test case is that all optimizations other than the optimizations listed in the cycle do not run when executed with the Alive InstCombine suite. In some cases, the test case generated for a sequence of optimizations in a cycle will not result in compiler non-termination when run with a suite of InstCombine optimizations. In those cases, an optimization not in the cycle has matched an input program intended for an optimization in the cycle, effectively breaking the cycle. This occurs if an optimization in the cycle is shadowed by another optimization. We have extended our analyses to check whether an optimization shadows another. Figure 14 provides an example of shadowing. We evaluate the number of cycles that are shadowed in our experimental evaluation.

5. Experiments

In this section, we describe and experimentally evaluate the prototype termination checker for Alive InstCombine optimizations. The goal of this evaluation is to show that (1) optimization sequences that cause non-termination are common in the Alive InstCombine suite and the termination checker detects them, (2) the non-termination bugs can be demonstrated in LLVM with the test cases generated, and (3) parallelization speeds up the exploration of optimization sequences for non-termination bugs.

Optimization	n-cycles			
	1	2	3	4
AddSub 1	1	1	1	6
AddSub 2	1	1	1	6
AddSub 3		1	5	14
AddSub 4			1	4
AddSub 5	1	3	6	13
AddSub 6	1	3	5	10
AndOrXor 1		1	3	8
AndOrXor 2	1	1	6	24
AndOrXor 3				2
AndOrXor 4			1	9
AndOrXor 5	1	1	2	11
AndOrXor 6		1	2	15
AndOrXor 7		1	8	36
AndOrXor 8	1	2	6	10
AndOrXor 9		1	1	12
AndOrXor 10		1	8	38
AndOrXor 11				1
AndOrXor 12		1	3	24
AndOrXor 13	1	1	3	22
AndOrXor 14		1	3	24
AndOrXor 15		1	1	12
AndOrXor 16		2	5	24
MulDivRem 1	1	1		
MulDivRem 2	1	2	2	
MulDivRem 3	1	1		
MulDivRem 4	1	2	2	
MulDivRem 5	1	1		
MulDivRem 6	1	2	2	
MulDivRem 7	1	1		
MulDivRem 8	1	2	2	
MulDivRem 9	1	2	3	7
MulDivRem 10		1	6	18
Select 1	1			
Select 2	1			
Shift 1	1	5	8	3
Shift 2	1	5	8	3
Shift 3	1	5		8
Shift 4	1			

Table 2: The optimizations from the InstCombine suite and the number of distinct n-cycles they participate in. The optimizations are named based on the InstCombine sources files where they occur in LLVM.

```
Name: AndOrXor 2
%op = or %X, C1
%r = and %op, C2
=>
%o = or %X, (C1 & C2)
%r = and %o, C2
```

(a)

```
Name: AndOrXor 5
Pre: C2&(-1 u>> C1)!=-1 u>> C1
%op = lshr %X, C1
%r = and %op, C2
=>
%op = lshr %X, C1
%r = and %op, C2 & (-1 u>> C1)
```

(b)

```
Name: AndOrXor 13
%op0 = or %A, C1
%r = or %op0, %op1
=>
%i = or %A, %op1
%r = or %i, C1
```

(c)

```
Name: AndOrXor 8
Pre: MVIZ(%A, -1 u>> CLZ(C))
%lhs = sub %A, %B
%r = and %lhs, C
=>
%neg = sub 0, %B
%r = and %neg, C
```

(d)

```
Name: Select 1
%c = icmp eq %X, C
%r = select i1 %c,%X,%Y
=>
%c = icmp eq %X, C
%r = select i1 %c, C, %Y
```

(e)

```
Name: Select 2
%c = icmp ne %X, C
%r = select i1 %c, %Y, %X
=>
%c = icmp ne %X, C
%r = select i1 %c, %Y, C
```

(f)

Figure 12: A sampling of the optimizations that cause 1-cycles. The preconditions in the optimizations are weak, which causes non-termination errors. The optimization (d) uses results of two dataflow analyses: MVIZ (MaskedValueIsZero) and CLZ (CountLeadingZeros).

5.1 Alive Termination Checker Prototype

The termination checker uses the publicly available version of Alive [25] as its foundation. The code generator and optimizations are compatible with the InstCombine pass of LLVM-3.6. We extended Alive to relax some of the typing restrictions to increase the expressivity of optimizations. Alive has rudimentary or no support for memory-related optimizations, `getelementptr`, and floating point optimizations. We excluded such optimizations for checking non-termination bugs. In total, we used 416 optimizations in the Alive InstCombine suite to generate optimization sequences for checking non-termination.

The test cases to demonstrate cycles were generated for LLVM-3.6 with Alive-generated code inserted into the InstCombine pass. We disabled constant folding in LLVM because our test cases use concrete constants for the optimizations that use dataflow analyses as described in Section 4. We use the unstable branch of Z3 [9], which has better support for quantifiers, for checking the constraints generated during cycle detection, type checking, and test-case generation. The Alive non-termination checker is about two thousand lines of python code and is available as open source[1].

Methodology. We will use the term *n-cycle* for an optimization sequence of length n that causes compiler non-termination. As discussed in Section 3.1, we restrict ourselves to examining simple n-cycles where each optimization appears at most once. There are $\frac{m!}{(m-n)!n}$ possible simple n-cycles for a suite of m optimizations. We cover all possible optimization sequences for small values of n. However, the state space increases quickly for larger values of n. We perform memoization to prevent exploring the same optimization multiple times with large state spaces. For example, when generating the composed optimization for the sequence $O_1O_2O_3$, we compose O_1 and O_2 into O_1O_2, memoize this composition and later reuse this composition for all sequences starting with the prefix O_1O_2. For larger values of n, we randomly sample selected sequences to find a million distinct compositions. We generate test cases for the detected cycles, and process them with a version of LLVM using Alive-generated C++ code for the InstCombine optimizations. All experiments were performed on a 64-bit Intel Haswell machine with four cores and 16 GB of RAM.

[1] `https://github.com/rutgers-apl/alive-loops`

Parallelization of optimization sequence exploration. The termination checker creates millions of Alive optimizations in the course of its search. To speed up this process, we built a parallel version of the non-termination detector that splits the checker into multiple processes with a master-slave architecture. A manager process divides the list of sequences to be explored into chunks that share a common prefix. Workers process chunks until they perform a predefined amount of activity and terminate. The manager creates new workers based on the amount of work that still needs to be performed.

5.2 Effectiveness in Detecting Cycles

Our prototype was effective in detecting optimization sequences that cause compiler non-termination. It detected 184 distinct optimization sequences that can cause non-termination. Table 1 also reports the number of optimization sequences explored, number of complete compositions, and the number of self-compositions possible. The number of feasible optimization sequences increase rapidly with the cycle length. We performed complete exploration of the state space for small cycle lengths (≤ 4). For larger cycle lengths, we performed exploration of random optimization sequences of length n until we were able to create one million complete compositions. Although there are a larger number of optimization sequences, a small number of them can be composed with each other. Even fewer of them produce a non-increasing source.

Analysis of cycles found. Table 2 reports the optimizations in InstCombine that participate in cycles and the number of distinct n-cycles that they appear in. A sampling of the 1-cycles and the 2-cycles that we discovered with our prototype is presented in Figure 12 and Figure 13 respectively. Detailed information about each optimization is available online.[2] There are 38 distinct optimizations that participate in 184 cycles.

[2] `https://github.com/rutgers-apl/alive-loops/blob/master/problems.opt`

(a)	(b)

`Name: AndOrXor 9`
`%op0 = xor %nOp0, -1`
`%op1 = xor %nOp1, -1`
`%r = and %op0, %op1`
`=>`
`%or = or %nOp0, %nOp1`
`%r = xor %or, -1`
 	
`Name: AndOrXor 15`
`%op0 = or %x, %y`
`%r = xor %op0, -1`
`=>`
`%nx = xor %x, -1`
`%ny = xor %y, -1`
`%r = and %nx, %ny`
	`Name: AndOrXor 12`
	`%na = xor %A, -1`
	`%nb = xor %B, -1`
	`%r = or %na, %nb`
	`=>`
	`%a = and %A, %B`
	`%r = xor %a, -1`
	
	`Name: AndOrXor 14`
	`%op0 = and %x, %y`
	`%r = xor %op0, -1`
	`=>`
	`%nx = xor %x, -1`
	`%ny = xor %y, -1`
	`%r = or %nx, %ny`

Figure 13: A sampling of the optimizations that cause 2-cycles. Optimizations A and B in Figure 4 also cause a 2-cycle
.

A weak precondition in the optimization is the main reason for non-termination errors in the majority of the 184 cycles. The optimization in Figure 12(a) will not cause non-termination if the precondition is strengthened to `C1!=(C1&C2)`. Similarly, the optimization in Figure 12(e) will not cause non-termination if the precondition ensures that the input variable `%X` is not a constant. Another significant fraction of the cycles involved optimizations that introduced instruction attributes when the instruction already had those attributes. Compiler writers typically write weak preconditions, which likely enables their optimization to run often. Our prototype checker detects non-termination errors in such scenarios and prevents them from getting into the tool chain.

Many of these optimizations participate in multiple cycles as shown in Table 2. Further investigation revealed that many of the longer cycles consisted of multiple smaller cycles. We found that majority of the 3 and 4-cycles were various different combinations of 1 and 2-cycles. After isolating the smaller cycles from the longer cycles, we were able to identify 32 distinct cycles, which do not contain any smaller cycles in them. Based on these observations and the difficulty in generating feasible complete compositions of an optimization sequence (see Table 1), we hypothesize that majority of the non-termination bugs can be discovered by exploring smaller cycles.

5.3 Demonstration of Errors with Test Cases

To enable the compiler writer to debug the 184 cycles, our prototype generated concrete inputs in the LLVM IR format for each of these cycles. When the generated test cases were compiled with LLVM using an Alive-generated InstCombine, the compiler would not terminate for 179 out of the 184 cycles. The remaining 5 cases were not able to induce compiler non-termination because the optimizations in the cycle were shadowed. In these cases, there was an optimization in the InstCombine suite that ran before the optimizations in the cycle, which disabled the cycle. The optimization in Figure 14(a) is a 1-cycle when C is `INT_MIN` (*i.e.*, minimum signed integer for a given bitwidth) and its self composition is shown in Figure 14(b). However, the optimization in Figure 14(a) is shadowed by the optimization in Figure 14(d) for the input program shown in Figure 14(c).

5.4 Execution time with Parallelization

Figure 15 presents the speedup with the parallelized versions of the termination checker when compared to the sequential version. The total execution time for sequential complete exploration of cycles ranges from 16 seconds (for $n = 1$) to 24 hours ($n = 4$). We were not able to run sequential versions for cycle lengths greater than 4. The parallel speedups are $1.15\times$ for exploring 1-cycles and $3.77\times$ for exploring 4-cycles on a 4-core machine. The speedups with the 1-cycle are lower because the exploration has relatively little work. The speedups are less than $4\times$ for execution on four cores while exploring larger cycles due to multiprocess communication overhead between the master and the workers and the additional parsing work performed by each worker thread. The parallelized version attains almost linear speedups with the increase in the number of cores. The parallelized version also enables exploration of cycles for higher cycle lengths.

6. Threats to Validity

The termination checker is built on top of Alive, which models the semantics of the LLVM IR. The semantics of the LLVM IR can change. Hence, the composition stage in our termination checker will likely be impacted by the discrepancies between the LLVM IR and the Alive semantics.

The termination checker tries to accurately capture the structure and the fixed point computation of InstCombine optimizations in LLVM. The infrastructure may need small modifications if InstCombine uses a different structure for its peephole optimizations, which can change the results.

The Alive suite is a snapshot of the InstCombine suite that has been aggregated over a period of time. We noticed that developers have strengthened preconditions in many optimizations in the current production release of LLVM in contrast to the Alive suite. Hence, the cycles reported probably may not occur in the production releases of LLVM. Although we focused on cycle detection for existing optimizations, the ideal use case for our termination checker is during the development of new optimizations especially with the interest in using Alive generated C++ code.

7. Related Work

We classify related prior research into following categories: (1) random testing for compiler bugs, (2) correct compilation, (3) termination checking for general purpose programs, and (4) performance bug identification.

Random testing. Testing with randomly generated code is one way to discover compiler non-termination errors [2, 19, 24, 28, 38]. Random testing has been effective in finding compiler errors. However, it is unlikely to discover corner cases that occur with rare inputs (*e.g.*, the optimization in Figure 14(a) will only cause a loop when the constant `C` has a specific value).

Correct compilation. Several domain-specific languages have been proposed for developing compiler optimizations [4, 14, 18, 20, 21, 36]. Prior approaches have typically focused on verifying individual optimizations. They do not address non-termination when a collection of optimizations are run until a fixed point. Superoptimizers [3, 16, 27, 33] that generate the shortest possible program for a particular code input typically avoid non-termination with cost metrics. However, these metrics are not directly applicable to InstCombine as it is both an optimization pass and a code normalization pass.

Translation validators [29, 31, 32, 40] check whether the compilation of a given input program is correct. Translation validators need the output of the compiler to check correctness, which is not available when the compiler does not terminate. Alternatively, if a verified compiler (*e.g.*, CompCert [22], Vellvm [39]) is written completely in a proof assistant such as Coq [8], then compiler termination is ensured by the proof assistant.

Termination checking. Detecting termination has been widely explored for a wide range of use cases such as imperative programs, term-rewriting systems, specifications of systems, and sys-

```
Pre: C < 0 &&
     isPowerOf2(abs(C))
  %p = sub %Y, %X
  %r = mul %p, C
=>
  %q = sub %X, %Y
  %r = mul %q, abs(C)
```

(a) Optimization is a 1-cycle

```
Pre: C < 0 &&
     isPowerOf2(abs(C)) &&
     abs(C) < 0 &&
     isPowerOf2(abs(abs(C)))
  %p = sub %Y, %X
  %r = mul %p, C
=>
  %q = sub %X, %Y
  %r = mul %q, abs(abs(C))
```

(b) Self-composition of (a)

```
definite i4 foo(i4 %Y,
                i4 %X) {
entry:
  %p = sub i4 %Y, %X
  %r = mul i4 %p, 8
  ret i4 %r
}
```

(c) Generated test case

```
Pre: isPowerOf2(C1)
  %r = mul %x, C1
=>
  %r = shl %x, log2(C1)
```

(d) Optimization shadowing (a)

Figure 14: The 1-cycle in (a) is shadowed by the optimization in (d) for the input shown in (c). The optimization (d) appears earlier than optimization (a) in the Alive InstCombine suite. Note that the condition `abs(C) < 0` is satisfied when `C` is `INT_MIN`.

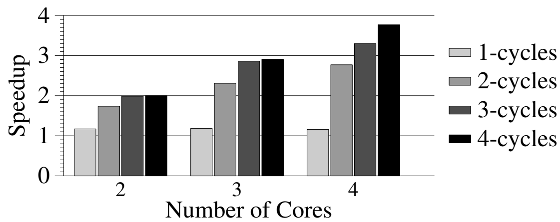

Figure 15: Execution time speedups with parallel exploration of n-cycles compared to the sequential exploration with increasing core count.

tems code [5–7, 13, 17, 23, 34, 34]. These techniques attempt to identify invariants either statically or dynamically that can be used to prove termination (*e.g.*, ranking function for a loop). Alive-generated C++ code can probably be analyzed with these systems. Identifying ranking functions for such code is likely not feasible because it also involves the LLVM infrastructure code.

More specifically, LLVM optimizations can be seen as a form of term rewriting systems. There is extensive research for showing termination and non-termination for term-rewriting systems [11, 12, 35, 37]. In contrast, our proposed approach leverages the structure and domain-specific knowledge of Alive optimizations to detect non-termination.

Performance bugs. Alive-generated code could be configured to stop operating on a basic block once the number of optimizations performed on it exceeds some threshold. In such a scenario, the compiler will terminate but with poor compilation times. Further, the generated code will likely have poor performance. There is active research on detecting the causes of poor performance [10, 15, 30]. These techniques are dynamic analyses that require a concrete input, which demonstrates a cycle. In contrast, the proposed termination checker detects these non-termination errors statically and also generates inputs that demonstrate cycles.

8. Conclusion

We have shown that non-termination bugs occur with peephole optimizations executed to a fixed point, especially when compiler developers are not careful with preconditions. Our methodology for detecting non-termination is based on composition of optimizations. We identified non-increasing source in self-compositions as a necessary condition for non-termination, and generated inputs to demonstrate non-termination with LLVM. Our goal was to create a tool that LLVM developers can use to check non-termination before they commit a new peephole optimization. Although we describe the methodology in the context of LLVM, it can be extended to peephole optimization frameworks of other compilers.

Acknowledgments

We thank Jay Lim, Adarsh Yoga, Vinod Ganapathy, and the ICSE reviewers for their feedback. This paper is based on work supported in part by NSF CAREER Award CCF–1453086, a sub-contract of NSF Award CNS–1116682, a NSF Award CNS–1441724, a Google Faculty Award, and gifts from Intel Corporation.

References

[1] LLVM PatternMatch. http://llvm.org/docs/doxygen/html/PatternMatch_8h.html. Retrieved 2016-02-12.

[2] A. Balestrat. CCG: A random C code generator. https://github.com/Merkil/ccg/. Retrieved 2016-02-12.

[3] S. Bansal and A. Aiken. Automatic generation of peephole superoptimizers. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 394–403, 2006.

[4] S. Buchwald. Optgen: A generator for local optimizations. In *Proceedings of the 24th International Conference on Compiler Construction (CC)*, pages 171–189, 2015.

[5] J. Burnim, N. Jalbert, C. Stergiou, and K. Sen. Looper: Lightweight detection of infinite loops at runtime. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 161–169, 2009.

[6] M. Carbin, S. Misailovic, M. Kling, and M. C. Rinard. Detecting and escaping infinite loops with Jolt. In *Proceedings of the 25th European conference on Object-oriented programming (ECOOP)*, pages 609–633, 2011.

[7] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 415–426, 2006.

[8] The Coq Development Team. *The Coq Proof Assistant Reference Manual*, 2013.

[9] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340, 2008.

[10] D. C. D'Elia, C. Demetrescu, and I. Finocchi. Mining hot calling contexts in small space. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 516–527, 2011.

[11] N. Dershowitz. Termination of rewriting. *Journal of symbolic computation*, 3(1):69–115, 1987.

[12] J. Giesl, P. Schneider-kamp, and R. Thiemann. AProVE 1.2: Automatic termination proofs in the dependency pair framework. In *Proceedings of the 3rd International Joint Conference on Automated Reasoning (IJCAR)*, pages 281–286, 2006.

[13] A. Gupta, T. Henzinger, R. Majumdar, A. Rybalchenko, and R.-G. Xu. Proving non-termination. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 147–158, 2008.

[14] S. Z. Guyer and C. Lin. Broadway: A compiler for exploiting the domain-specific semantics of software libraries. *Proceedings of the IEEE*, 93(2):342–357, Feb. 2005.

[15] S. Han, Y. Dang, S. Ge, D. Zhang, and T. Xie. Performance debugging in the large via mining millions of stack traces. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 145–155, 2012.

[16] R. Joshi, G. Nelson, and Y. Zhou. Denali: A practical algorithm for generating optimal code. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(6):967–989, Nov. 2006.

[17] C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, death, and the critical transition: Detecting liveness bugs in systems code. In *Proceedings of the 4th USENIX Symposium on Networked Systems Design & Implementation (NDSI)*, pages 243–256, 2007.

[18] S. Kundu, Z. Tatlock, and S. Lerner. Proving optimizations correct using parameterized program equivalence. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 327–337, 2009.

[19] V. Le, M. Afshari, and Z. Su. Compiler validation via equivalence modulo inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 216–226, 2014.

[20] S. Lerner, T. Millstein, and C. Chambers. Automatically proving the correctness of compiler optimizations. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)*, pages 220–231, 2003.

[21] S. Lerner, T. Millstein, E. Rice, and C. Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 364–377, 2005.

[22] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, July 2009.

[23] P. Li and J. Regehr. T-check: Bug finding for sensor networks. In *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, pages 174–185, 2010.

[24] C. Lidbury, A. Lascu, N. Chong, and A. F. Donaldson. Many-core compiler fuzzing. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 65–76, 2015.

[25] N. Lopes, D. Menendez, S. Nagarakatte, and J. Regehr. Alive: Automatic LLVM InstCombine Verifier. `http://github.com/nunoplopes/alive`. Retrieved 2016-02-12.

[26] N. Lopes, D. Menendez, S. Nagarakatte, and J. Regehr. Provably correct peephole optimizations with Alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 22–32, 2015.

[27] H. Massalin. Superoptimizer: A look at the smallest program. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 122–126, 1987.

[28] R. Morisset, P. Pawan, and F. Z. Nardelli. Compiler testing via a theory of sound optimisations in the C11/C++11 memory model. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 187–196, 2013.

[29] G. C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI)*, pages 83–94, 2000.

[30] A. Nistor, P.-C. Chang, C. Radoi, and S. Lu. CARAMEL: Detecting and fixing performance problems that have non-intrusive fixes. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, pages 902–912, 2015.

[31] M. Rinard. Credible compilation. Technical Report MIT-LCS-TR-776, Massachusetts Institute of Technology, Mar. 1999.

[32] H. Samet. Proving the correctness of heuristically optimized code. *Communications of the ACM*, 21(7):570–582, July 1978.

[33] E. Schkufza, R. Sharma, and A. Aiken. Stochastic superoptimization. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 305–316, 2013.

[34] F. Spoto, F. Mesnard, and É. Payet. A termination analyzer for Java bytecode based on path-length. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 32(3):8:1–8:70, Mar. 2010.

[35] J. Steinbach. Simplification orderings: History of results. *Fundamenta Informaticae*, 24(1–2):47–87, Apr. 1995.

[36] D. L. Whitfield and M. L. Soffa. An approach for exploring code improving transformations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(6):1053–1084, Nov. 1997.

[37] H. Xi. Towards automated termination proofs through “freezing”. In *Rewriting Techniques and Applications*, pages 271–285. Springer, 1998.

[38] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 283–294, 2011.

[39] J. Zhao, S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 427–440, 2012.

[40] L. Zuck, A. Pnueli, Y. Fang, and B. Goldberg. VOC: A methodology for the translation validation of optimizing compilers. *Journal of Universal Computer Science*, 9(3):223–247, 2003.