Termination-Checking for LLVM Peephole Optimizations

David Menendez
Department of Computer Science
Rutgers University
davemm@cs.rutgers.edu

Santosh Nagarakatte
Department of Computer Science
Rutgers University
santosh.nagarakatte@cs.rutgers.edu

ABSTRACT

Any mainstream compiler (e.g., LLVM, GCC) contains a large number of peephole optimizations that perform algebraic simplifications 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 new class of bugs with peephole optimizations: non-termination bugs. When a suite of peephole optimizations is executed until a fixed point, an optimization which undoes the effect of other optimizations will result in non-terminating compilation. This paper (1) proposes a methodology to detect non-termination bugs with 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.

1. INTRODUCTION

Compilers translate source programs to multiple target architectures while preserving semantics. Modern mainstream 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 are critical components of most compilers, because they not only clean up code resulting from other optimizations but also canonicalize code, which enables other optimizations. Typically, the compiler developer observes a pattern which 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 and nightly builds to ensure that the optimization is operating correctly. The peephole optimizations in the LLVM compiler are performed by the InstCombine pass, which has more than a thousand such peephole optimizations.

Peephole optimizations are a common source of compiler bugs [21, 27, 40]. These optimizations are buggy because of corner cases, especially in the presence of undefined behavior. Let us consider the InstCombine optimization in LLVM that transforms input code of the form \( x << c \) into \( x / (d / 1 << c) \), where \( x \), \( c \), and \( d \) are all integers (e.g., \( \text{int} \) in C). This optimization is beneficial when \( c \) and \( d \) are constants because it eliminates an instruction in the generated code (i.e., the expression \( d / 1 << c \) is a compile time constant). At the outset, the optimization appears to be correct because the source value produced is \( x \cdot 2^c/d \) and the target value produced is \( x / (d / 2^c) \), which is equal to \( x \cdot 2^c/d \). However, when \( x = 0 \), \( c = 3 \), and \( d = 7 \), the original program produces 0 as the output whereas the transformed program produces a division by zero error. Hence, this InstCombine optimization is incorrect as the transformed code produces a behavior that is not present in the source (LLVM bug PR#21245). To ensure that such optimizations are correct, compiler developers need to develop a specification of the optimization, check whether the optimization refines the behaviors of the source, and ensure that the implementation is correct.

To address this problem of developing correct peephole optimizations, our recent research has developed Alive, a domain specific language to specify peephole optimizations in LLVM [27]. An Alive optimization is of the form \( \text{source} \Rightarrow \text{target} \) with an optional precondition. The optimization checks the input code for the pattern of the form \( \text{source} \) and replaces it with the \( \text{target} \). Optimizations expressed in Alive are automatically checked for correctness using Satisfiability Modulo Theories (SMT) solvers. Alive abstracts various kinds of undefined behavior while encoding them appropriately for verification. 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 detected bugs in existing InstCombine optimizations [27], prevented bugs from getting into the LLVM repository [4], and the verification engine is actively used by LLVM developers. There are plans to replace the current InstCombine optimization pass in LLVM with Alive generated C++ code.

This paper describes a new class of non-termination bugs with peephole optimizations, which we discovered while translating InstCombine optimizations in LLVM to Alive. The LLVM compiler runs a suite of InstCombine optimizations multiple times until a fixed point (e.g., optimization level 0 runs the entire suite of InstCombine optimizations 7 times). Figure 2 provides a high level overview of the execution of InstCombine optimizations.

We observed that certain InstCombine optimizations can undo...
Our approach explores whether sequences of optimizations from InstCombine optimization. The proposed methodology extends Alive's suite of InstCombine optimizations. We have demonstrated that these optimizations cause the generated C++ code for InstCombine to loop indefinitely.

Contributions. This paper:
- Identifies a new class of bugs that cause compiler non-termination, which results due to 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. The proposed technique has discovered 184 optimization sequences, involving 38 optimizations, that cause non-termination in LLVM.

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

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

Figure 1: Workflow of our termination checking algorithm.

Figure 2: 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.

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 the LLVM's InstCombine optimizations. Hence, these non-termination bugs 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. We describe the methodology in the context of Alive because it already checks the correctness of an individual InstCombine optimization. The proposed methodology extends Alive to provide a global view of InstCombine optimizations in LLVM. Our approach explores whether sequences of optimizations from the Alive suite can have potentially bad interactions, which cause non-termination bugs when run until a fixed point. Figure 1 illustrates the methodology for detecting compiler non-termination bugs in InstCombine.

Our proposed methodology is based on the composition of a sequence of Alive optimizations. Given a suite of optimizations and a sequence length, the candidate sequence generation enumerates all possible ways in which the optimizations in the suite may be invoked and generates a sequence of the specified length. Once an optimization sequence is generated, the compose phase checks if it is feasible to compose the sequence of transformations, and generates a resultant transformation which summarizes the effect of running all the constituent transformations in the sequence one after the other.

An InstCombine optimization executed until a fixed point will terminate when the optimization composed with itself consumes a larger source input and transforms it into target code. If the optimization can be composed with itself and does not consume a larger source input than the single invocation of the optimization, we can run this optimization infinitely many times for a finite source input. Hence, the necessary conditions for the non-termination bugs 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, if the candidate sequence generator generates an optimization sequence \(O_1O_2O_3\), 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 \(O_z\) is satisfiable and the number of instructions in the source of \(O_z\) 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 a compiler non-termination bug.

When an optimization sequence can cause a non-termination bug, 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 bugs that cause compiler non-termination, which results due to 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. The proposed technique has discovered 184 optimization sequences, involving 38 optimizations, that cause non-termination in LLVM with Alive generated C++ code.
2. BACKGROUND ON ALIVE

Alive is a language for specifying peephole optimizations in LLVM. The Alive interpreter automatically checks the correctness of the optimization using an SMT solver and generates C++ code for use in LLVM, which implements the optimization. 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 optimization is parametric over types and bit widths. Hence, the Alive interpreter checks the correctness of the optimization for all feasible types and bit widths. Alive abstracts the various kinds of undefined behavior while the interpreter reasons about them during verification.

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.

2.1 InstCombine Optimizations in Alive

Alive optimizations are specified as source ⇒ 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 (\(i.e., \text{C2} = \sim \text{C1}\), where \text{C1} and \text{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 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, \text{C1} and \text{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 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 [27]. 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 the instruction specifies when an 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 [27]. Examples of such attributes are nsw (no signed wrap), nzw (no unsigned wrap), and exact. An arithmetic instruction with the no signed wrap attribute produces a poison value on signed overflows [27]. Poison values produce undefined behavior when such values are used in instructions with side effects. The poison value propagates with dependences. 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 instantiates candidate types for the optimization using the typing constraints of the instruction with the help of a SMT solver. 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: (1) whether the precondition is satisfiable, (2) whether the target is well-defined when the source is well-defined and poison-free, (3) whether the target is poison-free when the source is well-defined and poison-free, and (4) whether the values of the roots of the DAGs in the source and target produce the same values when the source is well-defined and poison-free. 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 [3]. 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 BUG DETECTION

Alive proves 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 run until a fixed point, the compiler will not terminate. Consider the two optimizations (O₁ and O₂) 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 to the InstCombine suite, it is necessary to determine if the newly proposed optimization can interfere with existing optimizations to avoid compiler non-termination bugs. LLVM InstCombine, which is a collection of C++ code, does not have such a global view. Over the years, LLVM InstCombine maintainers have experienced compiler non-termination bugs, which have been hard to debug, isolate and reproduce [2]. With Alive getting adoption from 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 2. Suppose, the optimization order in InstCombine is O₁O₂O₃. First, the optimization O₁ is attempted. If it is successful, then the newly created instruction is added to the work list and the entire suite of optimizations O₂O₃ is tried again as shown in Figure 2. If it is not possible to apply optimization O₁, then optimization O₂ is attempted as described earlier. If there exists an optimization O₄ which is applicable in a subset of the cases optimization O₁ is applicable, then optimization O₄ will never be invoked in the order O₁O₂O₃O₄ as consequence of this structure of InstCombine optimizations. We say that optimization O₄ shadows optimization O₁.

In the absence of shadowing, detecting non-termination bugs in InstCombine reduces to solving the following problem:

Given a suite of InstCombine optimizations where no optimization shadows the other, 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 (O₁O₂O₃ 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 (see Figure 4(d) and Figure 4(e)). We describe each of these steps in the following subsections. We will use the term loop and cycle to describe optimization sequences that cause compiler non-termination interchangeably.

3.1 Generating Candidate Sequences

Our goal is to determine whether there exists a sequence of optimizations that can loop/cycle 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!). Further, the space of optimization sequences with repetition is even larger. Hence, the candidate optimization sequence generation phase explores this large state space in a systematic fashion by iteratively exploring all sequences up to a certain length (i.e., we explore sequences of length 1, length 2, and so on).

In principle, a candidate sequence may be any sequence of Alive optimizations. However, we will often 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). We check each feasible composition of a candidate sequence to see whether it is a self-cycle.

In practice, we observed that we have to explore a large number of optimization sequences to find a feasible composition for cycle lengths greater than 6.

3.2 Composition

The basis of our procedure for detecting non-termination bugs is an optimization composition. When determining whether two optimizations O₁ and O₂ compose, we attempt to see if O₂ will match the result of applying O₁ to some input. Therefore, we treat the
function COMPOSE($O_1, O_2$)
    $Sets \leftarrow \text{AlignDAGs}(O_1, O_2)$
    $\Phi \leftarrow \emptyset$
    for all $S \in Sets$
        $\phi \leftarrow \text{SelectReplacement}(S)$
        $\Phi \leftarrow \Phi \cup \phi$
    end for
    $src(O_3) \leftarrow \text{Graft}(\text{root}(src(O_1)), Sets)$
    $tgt(O_3) \leftarrow \text{Graft}(\text{root}(tgt(O_2)), Sets)$
    $pre(O_3) \leftarrow \text{Graft}(pre(O_1), Sets) \wedge \text{Graft}(pre(O_2), Sets) \wedge \Phi$
    return $O_3$
end function

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

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.

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`, `%w` is 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.
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
            S_i ← S
        else
            S_i ← {t_i}
            Sets ← Sets ∪ {S_i}
        end if
    end for
    if S1 ≠ S2 then
        S ← 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(S) ← 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 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.

function MATCH(vp, v)
    if opcode(vp) ≠ opcode(v) then
        reject
    else if flags(vp) ⊈ flags(v) then
        reject
    else
        return [(op1(vp), op1(v)), (op2(vp), op2(v)), ...]
    end if
end function

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

for the operations on disjoint sets.

Figure 5 illustrates the process of DAG alignment. Initially, a tuple containing the roots (h1, h2) is added to the worklist. When (h1, h2) is processed, the instructions h1 and h2 are matched, which results in tuples (q1, q2) and ((C1 & C2), h2) being added to the worklist, and a set {h1, h2} is created. When (q1, q2) is processed, instructions q1 and q2 are matched, which results in (h1, h2) and (h2, h2) being added to the worklist, and a set {h1, h2} 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 (the CheckValidity function call in Figure 6). We check that any instruction in a set does not have another value in the same set as the operand either directly or transitively. We perform this check by detecting the cycles in the dependency graph constructed between the sets. The nodes of the dependency graph are sets from the DAG alignment and two nodes

function MERGE(v1, v2)
    if opcode(v1) ≠ opcode(v2) then
        reject
    else
        v3 ← copy(v1)
        flags(v3) ← flags(v1) ∪ flags(v2)
        return (v3, [op1(v1), op1(v2)), (op2(v1), op2(v2)), ...])
    end if
end function

Figure 9: Combine two patterns, if possible.

A and B have an edge if an instruction in a set A depends on the operand from set B. Such circular dependencies arise when the composition is not feasible. We reject such compositions.

We also ensure that any set from DAG alignment does not have both instructions and symbolic constants (e.g. C1) or constant expressions. We also check that any set does not contain two distinct specific constant literals (e.g., 1 and 2). When we are composing O1 with O2 with DAG alignment, we also check that any value in the source of O1 is not unified with an intermediate value generated in the target of O1. These checks ensure that the collection of sets generated from DAG alignment are well-formed and our algorithm generates only valid compositions according to Alive semantics.

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 O1’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 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 O1, we choose the value that occurs in the source of O1. If the set contains constant expressions other than the one selected (if any), 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, h2, C1 & C2} contains a symbolic constant, an input variable, and a constant expression. The symbolic constant C2 is present in the source of O1. Hence, the symbolic constant C2 is chosen as the replacement and we introduce new clauses to the pre-condition 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).

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 provides a fresh name to the new instructions or the symbolic constants that it creates. 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. Figure 5 presents the final optimization O3 generated with the graft procedure.
function \text{Graft}(t, \text{Sets}) 
\begin{align*}
\text{if} \ (\exists S \in \text{Sets} \text{ such that } t \in S \land replacement(S) \neq t) & \text{ then} \\
\text{return} \ \text{Graft}(replacement(S), \text{Sets}) \\
\text{else} & \\
\ t' & \leftarrow copy(t) \\
\text{for all operands } i \text{ do} & \\
\ op_i(t') & \leftarrow \text{Graft}(op_i(t), \text{Sets}) \\
\text{end for} & \\
\text{return} & t' \\
\end{align*}

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

\begin{align*}
\text{Pre: true} & \\
\ %p = \text{add} \ %a , \ %b & \Rightarrow \\
\ %r = \text{add} \ %p , \ %c & \Rightarrow \\
\ %q = \text{add} \ %b , \ %c \\
\ %r = \text{add} \ %a , \ %q \\
\text{Pre: true} & \\
\ %p = \text{add} \ %a , \ %b & \Rightarrow \\
\ %p1 = \text{add} \ %a1 , \ %b1 & \Rightarrow \\
\ %q = \text{add} \ %b , \ %c \\
\ %q1 = \text{add} \ %b1 , \ %c & \Rightarrow \\
\ %r = \text{add} \ %a1 , \ %q1 \\
\end{align*}

(a) | (b)

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.

3.3 Necessary Conditions 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, it is possible that an optimization can be self-composed with itself only a finite number of times. 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 optimization 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 result 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 is 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 inputs and return value corresponding to its root. If the root is \texttt{store}, then no value is returned. 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 \texttt{i8} as the type chosen for the source variables. The symbolic constants \texttt{C1} and \texttt{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 \( \texttt{WillNotOverflowSignedAdd}(\%a, \%b) \), 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 difficulty 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 \texttt{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 manifest in compiler
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.

5.1 Alive Termination Checker Prototype

The termination checker uses the publicly available version of Alive [1] 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 for demonstrating 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 [12], 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 open source.

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 \( \frac{(m - n)!}{m!} \) 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 \( O_1 O_2 O_3 \), we compose \( O_1 \) and \( O_2 \) into \( O_1 O_2 \), memoize this composition and later reuse this composition for all sequences starting with the prefix \( O_1 O_2 \). For larger values of \( n \), we randomly sample selected sequences to find a million distinct compositions. We generate test cases as input to LLVM, which uses the Alive generated C++ code for InstCombine optimizations, for the detected cycles. All experiments were performed on a 64-bit Intel Haswell machine with four cores and 16GB of RAM.

Parallelization of optimization sequence exploration. The termination checker creates millions of Alive optimizations in the course of its search. We experienced space leaks in Z3 with long running searches of optimization sequences, where the memory allocated in Z3 for all Alive variables were not released when the optimization is no longer used. Hence, a single long-running process will eventually use all the memory on a machine.

To address this problem, we built a parallel version of Alive termination checker that splits the checker into multiple processes with a master-slave architecture. The parallelization not only addresses the space leak problem with Z3 but also allows parallel searches. A manager process divides the list of sequences to be explored into chunks, which share a common prefix. Workers process chunks until they perform a predefined amount of activity and terminate. The manager creates new workers based on 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 errors. Table 1 also reports the number of optimization sequences that were 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 (\( \leq 4 \)).

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 number of feasible optimization sequences until a million complete compositions were found.

<table>
<thead>
<tr>
<th>( n )</th>
<th>Optimization Sequences</th>
<th>Complete Compositions</th>
<th>Self-compositions</th>
<th>Non-increasing</th>
<th>Cycles Found</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>416</td>
<td>416</td>
<td>1</td>
<td>25</td>
<td>23</td>
</tr>
<tr>
<td>2</td>
<td>86320</td>
<td>7001</td>
<td>4</td>
<td>31</td>
<td>27</td>
</tr>
<tr>
<td>3</td>
<td>23824320</td>
<td>182678</td>
<td>96989</td>
<td>49</td>
<td>35</td>
</tr>
<tr>
<td>4</td>
<td>737958120</td>
<td>524634</td>
<td>2694291</td>
<td>152</td>
<td>99</td>
</tr>
<tr>
<td>5*</td>
<td>11319090295</td>
<td>1000000</td>
<td>463017</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>6*</td>
<td>9761368054</td>
<td>1000000</td>
<td>394794</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7*</td>
<td>47416216578</td>
<td>1000000</td>
<td>395638</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Total Number of Cycles: 184

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.

<table>
<thead>
<tr>
<th>Optimization</th>
<th>( n )-cycles</th>
<th>Optimization</th>
<th>( n )-cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddSub 1</td>
<td>1 2 3 4</td>
<td>AddOrXor 14</td>
<td>1 3 5 6</td>
</tr>
<tr>
<td>AddSub 2</td>
<td>1 1 6</td>
<td>AddOrXor 16</td>
<td>1 1 12</td>
</tr>
<tr>
<td>AddSub 3</td>
<td>1 5 14</td>
<td>MulDivRem 1</td>
<td>1</td>
</tr>
<tr>
<td>AddSub 4</td>
<td>1 1</td>
<td>MulDivRem 2</td>
<td>1 2 2</td>
</tr>
<tr>
<td>AddSub 5</td>
<td>1 3 6 13</td>
<td>MulDivRem 3</td>
<td>1</td>
</tr>
<tr>
<td>AddSub 6</td>
<td>1 3 5 10</td>
<td>MulDivRem 4</td>
<td>1 2</td>
</tr>
<tr>
<td>AddOrXor 1</td>
<td>1 3 8</td>
<td>MulDivRem 5</td>
<td>1</td>
</tr>
<tr>
<td>AddOrXor 2</td>
<td>1 1 6 24</td>
<td>MulDivRem 6</td>
<td>2 2 0</td>
</tr>
<tr>
<td>AddOrXor 3</td>
<td>1 2</td>
<td>MulDivRem 7</td>
<td>1</td>
</tr>
<tr>
<td>AddOrXor 4</td>
<td>1</td>
<td>MulDivRem 8</td>
<td>1 2</td>
</tr>
<tr>
<td>AddOrXor 5</td>
<td>1 1 2 11</td>
<td>MulDivRem 9</td>
<td>1 2 3</td>
</tr>
<tr>
<td>AddOrXor 6</td>
<td>1 2 15</td>
<td>MulDivRem 10</td>
<td>1 6 18</td>
</tr>
<tr>
<td>AddOrXor 7</td>
<td>1 8 36</td>
<td>Select 1</td>
<td>1</td>
</tr>
<tr>
<td>AddOrXor 8</td>
<td>1 2 6 10</td>
<td>Select 2</td>
<td>1</td>
</tr>
<tr>
<td>AddOrXor 9</td>
<td>1 1 12</td>
<td>Shift 1</td>
<td>1 5 8 3</td>
</tr>
<tr>
<td>AddOrXor 10</td>
<td>1 8 38</td>
<td>Shift 2</td>
<td>1 5 8 3</td>
</tr>
<tr>
<td>AddOrXor 11</td>
<td>1</td>
<td>Shift 3</td>
<td>1 5 8</td>
</tr>
<tr>
<td>AddOrXor 12</td>
<td>1 3 24</td>
<td>Shift 4</td>
<td>1</td>
</tr>
</tbody>
</table>
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 Alive generated InstCombine C++ code, 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 \( \text{INT_MIN} \) (i.e., minimum signed integer for a given bitwidth) and its self composition is shadowed by the optimization in Figure 14(d) for the input 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 (for \( n = 4 \)). We were not able to run sequential versions for cycle lengths greater than 4. The parallel speedups are 1.15 for exploring 4-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 small amount of 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 instructions in the LLVM IR. The semantics of the LLVM IR evolve rapidly. Hence, the composition stage in our termination checker will likely be impacted by the discrepancies between the Alive semantics and the LLVM IR.

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.

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 [5, 21, 26, 29, 40]. 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 1-4(a) will only cause a loop when the constant \( C \) has a specific value).

Correct compilation. Numerous domain specific languages have been proposed for developing compiler optimizations including peephole optimizations [7, 20, 22, 23, 37, 38]. 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 [6, 18, 28, 34] 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 [30, 32, 33, 42] 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 [24], Vellvm [41]) is written completely in a proof assistant such as Coq [11], then compiler termination is ensured by the proof assistant.

Non-termination checking. Termination checking has been widely explored for a wide range of use cases such as imperative programs, term-rewriting systems, specifications of systems, and systems code [8, 9, 10, 16, 19, 25, 35, 39]. 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 [14, 15, 36, 39]. In contrast, our proposed approach leverages the structure of Alive optimization. Further, the non-increasing source can be considered as our ranking function.

Performance bugs. Alive could be configured to stop operating on a basic block once the number of optimizations performed on it exceeds some threshold. This converts non-terminating behavior to poor compilation. The generated code will also have poor performance. There is promising research on detecting the causes of poor performance, such as examining call graphs [13, 17] or loop conditions [31]. 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 fixed point based peephole optimizations especially when compiler developers are not careful with the precondition. We designed a methodology for detecting non-termination 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 optimization. We believe we have accomplished this goal.
References


