Hydra: Generalizing Peephole Optimizations with Program Synthesis

MANASIJ MUKHERJEE, University of Utah, USA
JOHN REGEHR, University of Utah, USA

Optimizing compilers rely on peephole optimizations to simplify combinations of instructions and remove redundant instructions. Typically, a new peephole optimization is added when a compiler developer notices an optimization opportunity—a collection of dependent instructions that can be improved—and manually derives a more general rewrite rule that optimizes not only the original code, but also other, similar collections of instructions. In this paper, we present Hydra, a tool that automates the process of generalizing peephole optimizations using a collection of techniques centered on program synthesis. One of the most important problems we have solved is finding a version of each optimization that is independent of the bitwidths of the optimization's inputs (when this version exists). We show that Hydra can generalize 75% of the ungeneralized missed peephole optimizations that LLVM developers have posted to the LLVM project's issue tracker. All of Hydra's generalized peephole optimizations have been formally verified, and furthermore we can automatically turn them into C++ code that is suitable for inclusion in an LLVM pass.

CCS Concepts: • **Software and its engineering → Translator writing systems and compiler generators**; *Source code generation.*

Additional Key Words and Phrases: program synthesis, generalization, superoptimization, llvm, alive2, souper, hydra, peephole optimization

ACM Reference Format:
Manasij Mukherjee and John Regehr. 2024. Hydra: Generalizing Peephole Optimizations with Program Synthesis. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 120 (April 2024), 29 pages. https://doi.org/10.1145/3649837

1 INTRODUCTION

Peephole optimizers perform *local rewrites*: they identify inefficient combinations of instructions and transform them into instructions that perform the same job more cheaply, often without changing the program's control flow. For example, following a De Morgan law, $\neg A \wedge \neg B$ can be rewritten as $\neg(A \vee B)$, reducing the computation cost by one operation.

Peephole optimizers are ubiquitous in optimizing compilers, and yet they can be somewhat dissatisfying artifacts: rather than implementing a coherent algorithm, they are typically little more than a collection of pattern matchers. Each such pattern represents an independent opportunity to be incorrect or unprofitable, and moreover there is a very long tail of patterns that we might wish to optimize: a peephole optimizer can never considered to be finished. The LLVM compiler includes more than 40,000 lines of C++ code implementing peephole optimizations, and a fuzzing campaign found it to be the single buggiest component of this compiler [Yang et al. 2011].

Authors' addresses: Manasij Mukherjee, University of Utah, Salt Lake City, USA, manasij@cs.utah.edu; John Regehr, University of Utah, Salt Lake City, USA, regehr@cs.utah.edu.

2475-1421/2024/4-ART120
https://doi.org/10.1145/3649837

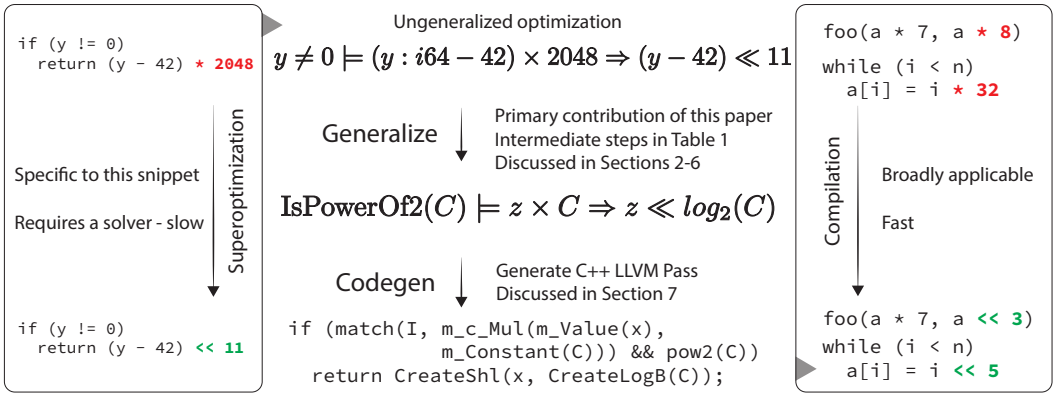

Fig. 1. Hydra transforms an ungeneralized optimization (top/left) from a developer or a superoptimizer into a generalized version that is often suitable for inclusion in a compiler (bottom/right)

The typical workflow for adding a new peephole optimization is:

(1) Someone notices a computation, either in optimized intermediate representation (IR) or in optimized object code, that could be rewritten to be more efficient.
(2) A compiler developer manually *generalizes* the rewrite, making it apply to more situations than just the single example that they originally intended to fix. Often this necessitates protecting the new optimization with precondition checks.
(3) The developer implements the generalized optimization in the compiler's implementation language.

After spending more than a decade watching compiler developers following this process, we have come to believe that step (2), the process of generalizing an optimization, is particularly error-prone. It should be automated, and supported by formal methods. This automation is the primary contribution of our paper, but we also automate step (3). Step (1), discovering unimplemented local rewrites, is not addressed by our work. Optimization discovery can be automated using a superoptimizer [Bansal and Aiken 2006; Sands 2011; Sasnauskas et al. 2017], and we have also found the LLVM project's issue tracker to be a fruitful source of desirable, ungeneralized optimizations. We created Hydra, a tool that generalizes these.

Let us look at a simple example of generalization. Assume that a developer notices code equivalent to $z \times 2048$ in a compiler's output. Since multiplying an integer by 2048 is (on many platforms) more efficiently expressed as a left shift, $z \ll 11$, this is a good candidate for a new peephole optimization. The compiler developer will readily observe that this pattern can be used to optimize multiplication by constants other than 2048; for example, $z \times 32$ should be optimized to $z \ll 5$. On the other hand, of course, there are many constants for which this pattern does not apply: $z \times 23$ cannot be implemented using any single shift instruction. Thus, the generalized optimization requires a precondition check; we write it as:

$$\mathrm{IsPowerOf2}(C) \models z \times C \Rightarrow z \ll \log_2(C)$$

Throughout this paper, we will refer to the three parts of a peephole optimization as the *precondition* (a predicate guarding the optimization, to the left of the $\models$), the *left-hand side* (a potentially optimizable collection of instructions, between the $\models$ and the $\Rightarrow$), and the *right-hand side* (a more efficient collection of instructions that refines the left-hand side when the precondition holds).

The example above illustrates another feature that is common in generalized optimizations: the optimized code contains a constant that is not found in the unoptimized code, but rather is a function of values found in the unoptimized code. To generalize an optimization, these functions must be discovered. To make this peephole optimization ready for inclusion in a real compiler, we might also wish to make it apply to the commuted version $2048 \times z$, and finally we would want it to apply to integer values of any bitwidth, not only to the width of the original, unoptimized code that someone noticed.

Bitwidth-independence ended up being a central concern; getting all of the parts to fit together cost us about a year in terms of getting Hydra working seamlessly. One interesting challenge that we faced was finding bitwidth-independent versions of optimizations whose preconditions involve the results of dataflow analyses. For example, if the compiler can prove that the bottom two bits of a value x are always set, then the expression $x \mid 3$ can be simplified to x. We introduce the notion of *dataflow variables* that make implementation-level elements of dataflow analyses available as synthesis components. For example, given a dataflow variable KnownBits(x).ones—a bitvector that contains a one in every bit position where x provably has a one in all executions—Hydra generalizes this optimization to:

$$C = (C \;\&\; \text{KnownBits}(x).\text{ones}) \models x \mid C \Rightarrow x$$

In other words, if the provable-ones in x are a superset of the ones in C, then the bitwise-or instruction is unnecessary.

> **Research goal:** We are creating basic technologies that will make it unnecessary for future compiler developers to implement peephole optimizations by hand. Rather, these optimizations will be automatically generated and automatically formally verified.

Our generalization toolchain (Sections 2–6) starts out with a peephole optimization expressed in Souper's [Sasnauskas et al. 2017] intermediate representation, which can be automatically extracted from a pair of LLVM functions exemplifying an optimization. Hydra's output, a generalized optimization, is then input to a code generator (Section 7), and the resulting code can be compiled into a shared library and loaded into LLVM 17 as an optimization pass; Figure 1 shows an example of how these pieces fit together. The generated code is correct by construction.

Our top-level result is that generalization works. For example, we took a collection of optimizations from Souper and used them to optimize the integer benchmarks from SPEC CPU 2017. The 10 optimizations that matched the most times fired a total of 18,695 times. We then ran the optimizations from Souper through Hydra and used its output to optimize the SPEC CPU 2017 integer benchmarks; in this case the top 10 optimizations matched a total of 442,860 times: a 23.7× increase. We also applied Hydra to ungeneralized optimizations that we found in the LLVM project's issue tracker; it was able to generalize 75% of these.

2 GENERALIZING PEEPHOLE OPTIMIZATIONS

This section gives an overview of our work and provides some context.

Hydra's approach to generalizing optimizations. Given two peephole optimizations, one of them is more general if its transformation can be applied in a proper superset of the situations in which the other one applies. Broadly speaking, our approach is greedy, attacking the generalization problem from different angles until no further improvements are possible. Table 1 shows an overview of this process, along with an example for each step.

Recall that a peephole optimization can be decomposed into three parts: Precondition $\models$ LHS $\Rightarrow$ RHS. The parts are interdependent and all of them must be modified during generalization. We start

Table 1. This example illustrates most generalization steps described in this paper, beginning with the sort of ungeneralized optimization that Souper typically emits. $y:\text{i64}$ indicates that y is a 64-bit integer and $z:\text{i4}$ indicates that z is a 4-bit integer; we omit redundant bitwidth constraints. Text in red indicates something that is going away in the next step; green indicates something the current step introduces. These steps are largely sequential, except for the two rows marked *Invalid*—it is usually necessary to try multiple options before we can return to a *Valid* state.

Step	Peephole Optimization	Valid?	Section
Original	$y \neq 0 \models (y:\text{i64} - 42) \times 2048 \Rightarrow (y - 42) \ll 11$	✓	Sec 2
Pruning	$z:\text{i64} \times 2048 \Rightarrow z \ll 11$	✓	Sec 3
Narrowing	$z:\text{i4} \times 2 \Rightarrow z \ll 1$	✓	Sec 3.2
Symbolic Constants	$z:\text{i4} \times C \Rightarrow z \ll D$	×	Sec 4
Expression Synthesis	$z:\text{i4} \times C \Rightarrow z \ll log_2(C)$	×	Sec 4
Precondition Synthesis	$\text{IsPowerOf2(C)} \models z:\text{i4} \times C \Rightarrow z \ll log_2(C)$	✓	Sec 5
Width Independence	$\text{IsPowerOf2(C)} \models z \times C \Rightarrow z \ll log_2(C)$	✓	Sec 6

with a pruning step that removes inessential elements from the optimization. Second, we try to rewrite the entire optimization to work at a narrow bitwidth; if successful, this significantly speeds up the numerous solver calls that we will subsequently need to make. Next, we try to express all remaining preconditions in terms of dataflow facts, which we weaken as much as possible. After that, we try to replace literals with synthesized constant expressions; this potentially requires additional preconditions to be synthesized. Finally, we attempt to prove that the optimization is bitwidth-independent.

Acquiring ungeneralized optimizations. Hydra's input—an ungeneralized peephole optimization—typically starts out as a concrete pair of unoptimized and optimized LLVM IR. For example, we have found a number of desirable, but unimplemented, optimizations described this way in the LLVM project's issue tracker. We then (automatically) convert these into Souper's IR. Alternatively, we can use Souper to extract a LHS from LLVM IR into its own IR, and then use Souper to synthesize a RHS. In either case, the optimization that we start with is completely specific: it applies to a particular collection of instructions, containing specific literal constants, at specific bitwidths.

Estimating profitability. An optimization only makes sense when it is *profitable*: when the RHS is cheaper than the LHS for some cost model of interest. Figuring out the right cost model for LLVM IR is not totally straightforward because LLVM's backends are quite sophisticated and perform numerous optimizations of their own. Therefore, we have gone with a simple code model: most instructions have cost one; the ternary "select" instruction and some simple intrinsics such as funnel shifts have cost three, and then more complex intrinsics such as "reverse bits" and "count leading zeroes," as well as the division and remainder instructions, have cost five. This is roughly compatible with the observed behavior of LLVM's existing peephole optimizations.

Scope. Our current Hydra implementation targets LLVM and uses Alive2 [Lopes et al. 2021] as a verification engine. We support control-flow-free DAGs of arbitrary-width integer operations in the static single assignment (SSA) form. The instructions involved in a peephole optimization do not need to be located near each other, and can even live in different basic blocks—they only need to be connected by dataflow edges. We have avoided supporting floating point operations because, in our experience, many non-trivial queries about them cause SMT solvers to timeout. We have not supported memory operations because we judged them to be orthogonal to most of the techniques that we wanted to explore.

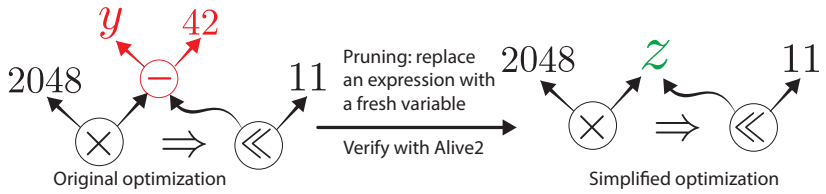

Fig. 2. Pruning an optimization. $(y - 42)$ is replaced with the fresh variable z that can match any expression. A subtraction of 42 is no longer necessary for the optimization to apply.

Applicability beyond LLVM. Although the details obviously differ, there is considerable overlapping functionality between the peephole optimizers in the various SSA-based compilers that we have looked at. Simplifications such as removing a pair of operators that undo each other, or applying a DeMorgan law, are ubiquitously useful. Even so, we must be extremely cautious when reusing peephole optimizations across compilers: details such as the undefined behavior model can invalidate apparently innocuous transformations. By reusing Alive2 as a black-box verification engine, Hydra's core remains independent of most of this kind of tricky, LLVM-specific reasoning. Our hypothesis is that if an Alive2-like tool existed for the internal representation of compilers such as MSVC, Intel CC, and GCC, then we could support generalizing peephole optimizations for these other compilers with relatively modest implementation effort. This hypothesis has not been tested, our current work remains specific to LLVM.

3 PREPROCESSING

This section describes two relatively lightweight steps that we take before moving on to proper generalization. The first one (pruning) is a simple generalization strategy on its own, and the second one (narrowing) makes generalization faster.

3.1 Pruning

Ungeneralized optimizations written by hand tend to not contain extraneous elements, but those emitted by tools can contain garbage that we want to eliminate prior to generalization. In particular, optimizations from Souper often contain irrelevant instructions and unnecessary preconditions. For preconditions, we greedily remove anything that can be removed without breaking the optimization. For instructions, we greedily replace them with fresh variables—when this results in an invalid optimization, we back out the change and try to remove a different instruction. Pruning results in an optimization that is more general than the original one, and is easier to generalize further. Consider the original optimization in Figure 2. It cannot optimize expressions like $(y - 43) \times 2048$, or $(a + b) \times 2048$, whereas the pruned version can.

We also try to remove LLVM's undefined-behavior-related flags from instructions, when this can be done without breaking the optimization. For example, if the LHS of an optimization contains an *add nsw* instruction (addition, with signed overflow being undefined) but this undefinedness is not necessary to drive the optimization, then we can remove this flag. Removing these flags gains generality: every optimization with an *add* on the LHS is also valid for *add nsw*. The converse does not hold.

3.2 Narrowing Bitwidth, to Make Generalization Faster

We can make generalization faster and more likely to succeed by performing it on an equivalent optimization where individual SSA values have a narrower bitwidth than they do in the original

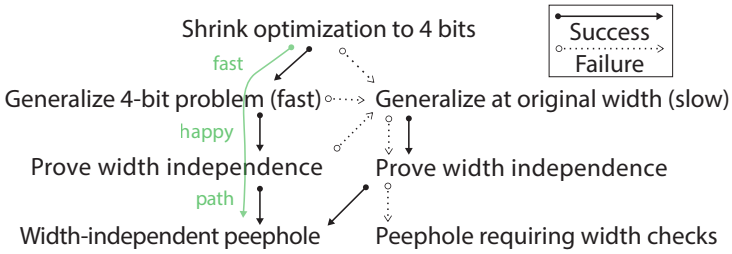

Fig. 3. Generalization pipeline with width reduction

optimization. Consider this ungeneralized optimization:[1]

$$x : \text{i64} \;/_u\; 2^{51} \Rightarrow x \gg_u 51$$

Generalizing this at its full width is challenging because wide divisions result in slow solver queries and timeouts. On the other hand, a narrower version of the same optimization, such as

$$x : \text{i4} \;/_u\; 2^2 \Rightarrow x \gg_u 2$$

would result in much faster queries, and few or no timeouts. Narrowing the width of an optimization is not always possible, but when it is, it can be a powerful technique. We define a narrow version of an optimization as one where at least one SSA value has a narrower bitwidth than it does in the original optimization and the LHS, RHS, and precondition expressions are structurally identical, differing only in the values of literal constants. If narrowing fails, we fall back to generalizing the original optimization as-is, at its native width. Figure 3 illustrates this augmented generalization workflow. Besides making solver invocations faster, narrowing the width of an optimization made it practical for us to utilize brute-force model counting for ranking synthesized preconditions (Section 4.3).

Type assignments. To create a narrow version of an optimization, we replace all SSA values with narrower versions of themselves. Although this is straightforward in many cases, it is more interesting when an optimization involves values with different widths. We fix the widths of the inputs to the optimization and then ask Alive2 to generate widths for internal nodes in the optimization, in such a way that the type system for LLVM IR is respected. This is a simple type system based on constraints that insist, for example, that both operands and also the result of a binary operator have the same width, that a sign-extend or zero-extend operation increases the width by at least one, and that the result of an integer comparison instruction is a Boolean value (i.e., it has a width of one). To these constraints we add two more:

- Do not exceed a configurable limit for any width
- Do not reduce the width of any variable to one (single-bit integers often behave quite differently than wider values)

Constant synthesis. Next, we synthesize fresh literal constants on both the LHS and RHS, at the narrowed width, under the constraint that the optimization still works. This is a standard exists-forall problem; Z3 can solve these natively but we have found that a custom CEGIS loop employing generalization by substitution [Dutertre 2015] performs better.

A problem that we observed in practice is that, sometimes, particular choices of synthesized constants can change the nature of the optimization. For example, if we end up synthesizing

[1]Throughout this paper we follow LLVM's convention of attaching signedness to operators instead of values. Thus, $/_u$ is unsigned division and $\gg_u$ is unsigned (logical) right shift; their signed equivalents are $/_s$ and $\gg_s$.

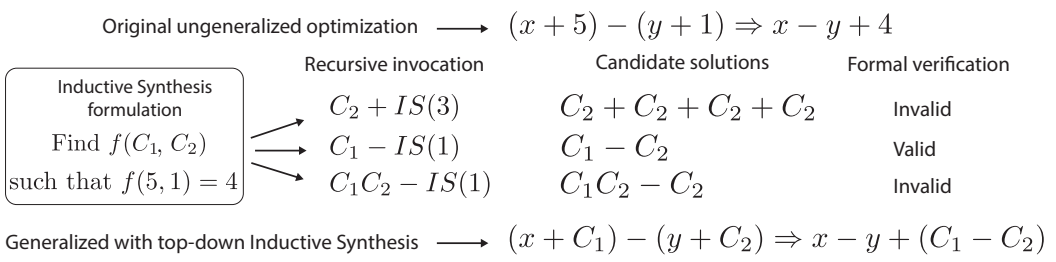

Fig. 4. Synthesizing expressions from input-output examples

addition by zero or multiplication by one, the resulting code trivially folds away. We addressed this issue by constraining constant synthesis to avoid the identity elements, if any, for the operations in the LHS and RHS.

4 MAKING CONSTANTS SYMBOLIC

One of the most important parts of generalization is weakening literal constants on the LHS to symbolic constants. For example, consider this (contrived) rewrite:[2]

$$(x + 5) - (y + 1) \Rightarrow x - y + 4$$

A more general form is

$$(x + C_1) - (y + C_2) \Rightarrow x - y + C_3$$

In some cases, replacing literal constants with symbolic constants will require protecting the optimization with one or more preconditions; we address this issue starting in Section 4.3 and then Section 5. Here, no precondition is required, but we need to figure out how to compute C_3 as a function of C_1, and C_2. We do this by solving a synthesis problem:

> Find an expression, using the standard arithmetic and bitwise operators, for computing C_3 in terms of C_1 and C_2, that makes the refinement relation $(x+C_1)-(y+C_2) \Rightarrow x-y+C_3$ hold.

The simplest satisfying expression is $C_3 = C_1 - C_2$. Substituting this expression, we get a complete, generalized optimization, that reduces the computation cost by one arithmetic operation:

$$(x + C_1) - (y + C_2) \Rightarrow x - y + (C_1 - C_2)$$

We call $C_1 - C_2$ a "constant expression." We generate constant expressions using two strategies: synthesis from input-output examples (which is roughly depth-first), and enumerative synthesis (which is roughly breadth-first). Our approach here is inspired by LENS [Phothilimthana et al. 2016], which employs multiple synthesis techniques based on the insight that—so far at least—no single synthesis technique appears to subsume all of the others.

4.1 Synthesizing Constant Expressions from Input-Output Examples

Synthesizing expressions from input-output examples—also known as inductive synthesis—is a well researched problem [Abate et al. 2018; Gulwani 2011; Jha and Seshia 2017]. In our case, we start with a single input-output example, created using the literal constants from the LHS and RHS of the ungeneralized optimization. This form of synthesis produces candidates that are guaranteed to be correct for the given input-output examples; a necessary but insufficient criterion for producing a

[2]Instead of employing this sort of peephole optimization, a production compiler would be more likely to reassociate the expression to $x - y + 5 - 1$ and then perform constant folding.

Algorithm 1 Top-down inductive synthesis assuming commutative binary operators.

```
1:  procedure Inductive-Synthesis(target, depth, context)
2:      if depth = 1 then                                  ▷ Base case. Fall back to enumeration.
3:          return Enumerative-Synthesis([expected=]target, [depth=]1)   ▷ Supports non-invertible operations.
4:      Results ← ∅
5:      for C ∈ Components do
6:          if target = Input-Examples[C] then             ▷ Is there a direct match?
7:              Results ←append— C
8:          for ⊙ ∈ Operators do
9:              if Detect-NOP(context, ⊙, C) then          ▷ Avoid expressions like C − C.
10:                 Continue
11:             x ← Solve-Equation(C ⊙ x = target)         ▷ Has to be fast. Can not use an SMT solver.
12:             Continue if x not found or guessed.        ▷ Typically fails for some non-invertible operations.
13:             for expr ∈ Inductive-Synthesis(x, depth − 1, {C, ⊙}) do
14:                 Results ←append— (C ⊙ expr)
15:     Return Results
```

valid optimization. Establishing sufficiency requires a potentially-expensive solver invocation, so we try to produce as few infeasible candidates as possible. This approach is illustrated in Algorithm 1. Figure 4 shows some expressions that produce the correct result for the given example, but do not satisfy the overall refinement relation.

We adopt a greedy top-down solution where we first choose a symbolic constant from the LHS and a top level operation $\odot$: $f(C_1, C_2, ...) = C_n \odot x$. We solve this equation to find a unique x whenever $\odot$ is invertible. For bitwise operations like *and*, *or*, and shifts, we try to find any one value that satisfies the equation—this is not guaranteed to work. We skip the non-invertible operations for which we fail to solve this equation. Next, we recursively invoke the same procedure with x as the output example. For the base case, we check if the new output example is equal to one of the input examples, or can be obtained by combining two input examples with a single operation—standard enumerative synthesis. This is helpful for non-invertible operations where we might fail to find a suitable x.

We prune away nop expressions such as $C_1+0, C_1\times(C_2/C_2)$ by building some identity-recognition rules into the operator selection logic, based on the last two tokens synthesized ("context," in Algorithm 1).

This avoids run-away chains like $C_3 = C_2 + n - n + n - n + \ldots$, because the first $(n - n)$ is blocked. Pruning is helpful, but it is insufficient to tackle the large search space resulting from redundant expressions. Hence, our search is bounded by a configurable depth limit, which restricts the total number of synthesized operations.

We have found that this synthesis technique is effective in finding expressions consisting of invertible arithmetic operations. State of the art synthesis tools such as FlashFill++ [Cambronero et al. 2023] confirm this observation; this paper mentions that putting *effectively invertible* operations closer to the root of the search tree is a key strategy for programming-by-example. Inductive synthesis is a fast and effective first step in our synthesis pipeline; it succeeds about 70% of the time; for the remaining 30%, we back off to enumeration.

4.2 Enumerative Synthesis, Sketching, and Components

Enumeration is a brute-force approach to synthesis. We have reused the building blocks of enumerative synthesis from Souper [Sasnauskas et al. 2017], augmenting it in two ways that make it more effective at solving the problems that we encounter.

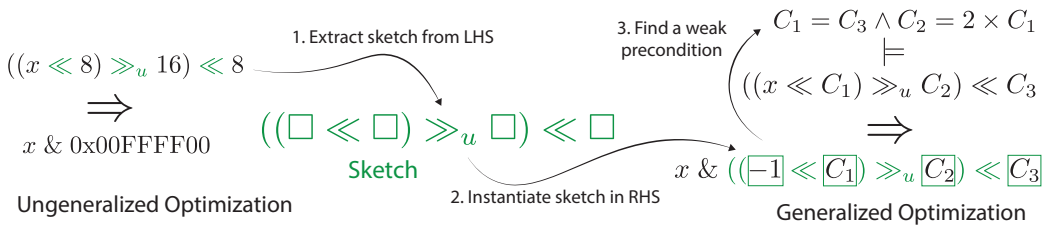

Fig. 5. Synthesizing expressions with sketching

First, we use sketching, where subexpressions from the LHS are used as templates for elements of the constant expression on the RHS. For example, consider this ungeneralized optimization:

$$((x : \text{i32} \ll 8) \gg_u 16) \ll 8 \Rightarrow x \ \& \ \text{0x00FFFF00}$$

Both GCC and LLVM perform this optimization, which reduces the computation cost from three operations to one. It can be generalized by making the constants symbolic and replacing 0x00FFFF00 with an expression. We can synthesize this expression using enumerative synthesis, but this is slow because the search space is large.

Sketching helps us to reduce the search space. Notice that the constant expression $((-1 \ll C_1) \gg_u C_2) \ll C_3$ is remarkably similar to the LHS computation $((x \ll C_1) \gg_u C_2) \ll C_3$. This happens fairly frequently. To make use of this sketch, we extract a function $f(A, B, C, D) = ((D \ll A) \gg_u B) \ll C$. Then, given the symbolic constants C_1, C_2, and C_3, we instantiate f with all $4^4 = 256$ combinations of the available LHS constants and three special constants $0, 1, -1$: $f(C_1, C_1, C_1, C_1)$, $f(C_1, C_1, C_1, C_2)$, etc. These are then used as components for enumerative synthesis. Additionally, this optimization has a precondition; we address synthesizing these in the following section. Figure 5 illustrates this process.

The second way that we improved Souper's synthesis algorithm is by adding a collection of components that are frequently useful in our domain. These include a log_2 component, a symbolic width component (explained in more detail in Section 6), and several auxiliary operators such as $C \oplus -1$; $-1 \ll C$; $-1 \gg_s C$; $-1 \gg_u C$; $\min(C_1, C_2)$; and, $\max(C_1, C_2)$. With these two improvements, we were able to synthesize constant expressions consisting of two new operations (not counting the ones in the sketch or the components) within a ten minute timeout. This was sufficient for most of the problems we encountered.

4.3 Synthesizing Preconditions Over Constants

It is frequently the case that an optimization only works for some choices of values for symbolic constants on the LHS. For example, continuing with the optimization from the previous section, its general form is:

$$C_1 = C_3 \wedge C_2 = 2 \times C_1 \models ((x \ll C_1) \gg_u C_2) \ll C_3 \Rightarrow x \ \& \ ((-1 \ll C_1) \gg_u C_2) \ll C_3$$

Generating preconditions over symbolic constants, such as $C_1 = C_3 \ \wedge \ C_2 = 2 \times C_1$ is a core problem solved by Alive-Infer [Menendez and Nagarakatte 2017]. Our approach is similar but simpler; it is comprised of three phases: precondition generation, filtering and verification, and combination. The goal is to synthesize a compact and weak precondition over symbolic constants that is sufficient to render the optimization sound.

Generating preconditions by enumerative synthesis. We start by generating preconditions that use the given symbolic constants; this is done by enumerative synthesis. The primary components for

enumeration are symbolic constants and the standard arithmetic and bitwise operators, and (similar to what we did in Section 4.2) we have augmented these components with sub-expressions from the LHS, after replacing literal constants with symbolic constants. We have experimented with using our inductive synthesis engine for synthesizing preconditions, but we found that enumerative synthesis was faster, and found weaker preconditions earlier in the search.

Precondition filtering and verification. We use one positive example and multiple negative examples to rapidly filter out expressions that are not valid preconditions. The positive example comes from the ungeneralized optimization itself, and the negative examples are obtained with fast solver queries on the partially generalized optimization. We use a counterexample-driven loop to obtain up to a configurable number of negative examples with solver queries. A valid precondition for constructing a generalized optimization must evaluate to false for all the negative examples, and true for the positive example. After filtering, a solver query is needed to verify that a candidate precondition is actually a precondition. We have observed that we need more than one negative example for effective filtering, but that the effectiveness has diminishing returns for more than three negative examples.

Ranking and combining preconditions. Given a collection of valid preconditions, we want to create a weak, compact disjunction of them. Alive-Infer [Menendez and Nagarakatte 2017] approaches this problem by generating more than one positive example with a solver, and then defining the weakest precondition as the one that evaluates to *true* for the largest number of positive examples. PSyCO [Lopes and Monteiro 2014] goes one step further and utilizes unsat cores to find the relevant subset of preconditions. We found that there are usually only a few candidates which pass through the filtering stage, so we have taken a heavier-weight approach: we rank preconditions using brute-force model counting when the total bitwidth of the symbolic constants is not greater than 12. This works well in conjunction with our bitwidth reduction strategy (Section 3.2). For larger widths (i.e. when width reduction fails), we fall back to a heuristic that smaller preconditions, which do not contain equality or constants, are weaker. Given a list of preconditions sorted in order of increasing strength, we greedily choose a subset of preconditions that imply the weakest precondition; Algorithm 2 shows how this works.

Algorithm 2 Combining preconditions to find a compact, weak precondition

```
1: procedure COMBINE-PRECONDITIONS(Preds)
2:     WP = false
3:     Rank Preds from weakest to strongest
4:     for P ∈ Preds do
5:         if ¬(WP ⟹ P) then                    ▷ Check with a solver
6:             WP ← WP ∨ P
7:     Return WP
```

5 GENERATING AND WEAKENING DATAFLOW PRECONDITIONS

LLVM—like other compilers—runs a collection of dataflow analyses that compute invariants over SSA values. These dataflow facts are the main way that it establishes preconditions over variables that are involved in peephole optimizations. This section describes our approach to establishing and weakening dataflow preconditions while generalizing optimizations.

5.1 Dataflow Preconditions

The major abstract domains that Hydra supports are:

- *Known bits*, which attempts to prove that individual bits are zero or one in all executions; for example, an 8-bit variable that is provably a multiple of four would be represented by *xxxxxx*00, where each *x* is a bit whose value is unknown;
- *Integer ranges*, which tries to prove that a given SSA value is within a range, such as [5..10], in all executions of the program being compiled; and,
- *Demanded bits*, which attempts to prove that some bits of an SSA value are not used by subsequent computations, for example because they are going to be masked off or truncated away.

Known bits and integer ranges are forward analyses, while demanded bits is a backwards analysis. We also support four single-bit forward dataflow analyses that LLVM uses to prove that an SSA value is respectively a power of two, non-zero, non-negative, and negative. We have focused on these for our prototype since they figure prominently in LLVM's peephole optimizations and also they are well-supported by Souper, which also uses them to prune its search space during synthesis [Mukherjee et al. 2020]. These analyses have been carefully formalized [Taneja et al. 2020], and tested in LLVM.

Consider this optimization that removes an instruction that sets the bottom bit of a variable, in the case where that bit is provably already set:

$$\text{KnownBits}(v) \sqsubseteq \text{xxxxxxx1} \models v \mid 1 \Rightarrow v$$

When this optimization is used by a compiler, checking its *concrete dataflow precondition* is a simple matter of invoking the partial order check for the known bits lattice. In other words, ensuring that the abstract value attached to the SSA value v implies the precondition holds. For example, if the compiler has proved that the bottom three bits of an 8-bit variable v are set, then its known bits representation would be *xxxxx*111, and this clearly implies that this optimization's precondition holds.

5.2 Dataflow Variables: Exposing Abstract Values as Synthesis Components

The optimization just above is a member of a large family of similar optimizations such as:

- $\text{KnownBits}(v) \sqsubseteq \text{xxxxxx1x} \models v \mid 2 \Rightarrow v$,
- $\text{KnownBits}(v) \sqsubseteq \text{xxxxx1xx} \models v \mid 4 \Rightarrow v$,
- $\text{KnownBits}(v) \sqsubseteq \text{xxx1xxx1} \models v \mid 17 \Rightarrow v$, and so on.

We want to collapse this collection of 2^{bitwidth} optimizations into a single, generalized optimization, but this cannot be accomplished using a partial order check against a specific abstract value such as xxxxxx1x. The problem is that the abstract value that we wish to compare against is a function of the constant value found on the LHS.

The generalized optimization that we wish to generate is something like:

$$\text{KnownBits}(v) \sqsubseteq \text{lift}_{\text{KB-Ones}}(C) \models v \mid C \Rightarrow v$$

where $\text{lift}_{\text{KB-Ones}}(C)$ is a function that lifts the one bits found in the literal constant C into an abstract value in the known bits domain, leaving the zeroes as unknown bits.

Although it would not be difficult to add components to our precondition synthesis that support lifting constants into abstract values, this by itself is not powerful enough to support all of the optimizations that we wish to generalize. Consider this transformation:

$$(\text{KnownBits}(v).\text{ones} \mid \text{KnownBits}(v).\text{zeroes} \mid \sim\text{DemandedBits}) = -1 \models v \Rightarrow \text{KnownBits}(v).\text{ones}$$

Here $\text{KnownBits}(v)$.ones is a bitvector containing a one for every bit that was provably one in the abstract value from the known bits domain (and zeroes elsewhere), $\text{KnownBits}(v)$.zeroes

contains a one for every bit that is provably zero in the abstract value (and zeroes elsewhere), and DemandedBits is a bit mask containing a one for every demanded bit of the root expression (v, in this case). The effect of this optimization is to replace an SSA value with a specific literal constant, which has ones corresponding to KnownBits(v).ones, and has zeroes elsewhere. This can be done when every bit of the original SSA value is either provably one, provably zero, or provably not-demanded.

The key insight permitting this optimization to be generalized from a specific example is that elements of abstract values must be exposed as synthesis components so that they can be used in arbitrary computations, including those that produce an RHS value. This concept, which we call *dataflow variables*, is implicitly present in the implementation of compilers such as LLVM and GCC, in the sense that these compilers contain optimizations that involve parts of abstract values in computations other than partial order checks.

We support the following dataflow variables:

- KnownBits(v).ones — v is guaranteed to contain a one in every bit position where this dataflow variable contains a one
- KnownBits(v).zeroes — v is guaranteed to contain a zero in every bit position where this dataflow variable contains a one
- IntegerRange(v).sMin — v is guaranteed to be at least as large as this dataflow variable, using signed comparison
- IntegerRange(v).sMax — v is guaranteed to be no larger than this dataflow variable, using signed comparison
- IntegerRange(v).uMin — v is guaranteed to be at least as large as this dataflow variable, using unsigned comparison
- IntegerRange(v).uMax — v is guaranteed to be no larger than this dataflow variable, using unsigned comparison
- DemandedBits — for every bit position where this dataflow variable contains a zero, the corresponding bit in the root of the LHS is guaranteed to be a don't-care: it can take either value without affecting the computation

Why these specific dataflow variables? We have two answers. First, some of these were suggested by reading existing LLVM optimization code, where we saw hand-written optimizations using something effectively equivalent to the known bits and demanded bits dataflow variables. Second, for integer ranges (which are not heavily used by LLVM's peephole optimizers, for reasons that are not entirely clear to us), the four dataflow variables listed here seemed intuitively likely to be useful. We could not, for example, find uses for dataflow variables that represent the midpoint of the range, or the width of the range. This aspect of our work remains more of an art than a science and we are still working to understand it better.

One might ask: Why is it legal to perform arbitrary computations on values that are derived from implementation-level aspects of dataflow facts? This works because, regardless of the presence of any dataflow variables, we fully formally verify each optimization under the assumption that its preconditions (including dataflow preconditions) hold. Then, separately, dataflow variables serve as additional synthesis components.

5.3 Normalizing Preconditions to Dataflow Preconditions

Hydra's inputs are ungeneralized peephole optimizations with preconditions that are expressed in three different ways. First, the explicit dataflow preconditions that we have already described. Second, implicit preconditions that are established by instructions that are part of the left-hand side of a peephole optimization, but that are not rewritten by the transformation. For example,

consider the expression $(x : \text{i8} \,\&\, 127) \,/_s\, 16$. Recall that $x : \text{i8}$ means that x is an 8-bit integer, $\&$ is the bitwise-and operator, and $/_s$ is the signed division operator. Although signed division cannot, in general, be reduced to a single shift operator, this specific example can be reduced to a right-shift by four bit positions because the dividend is non-negative: its sign bit has been cleared by the masking operation. The *and* instruction acts as an implicit precondition: the optimization does not work without it, but it is not otherwise involved in the transformation.

The third kind of precondition is the *path condition*—an arbitrary predicate over variables in the left-hand side. Path conditions are found in ungeneralized optimizations from Souper, which gets them by analyzing control flow in the program being optimized. For example, consider this C code, where `x` is a signed integer:

```
if (x < 0) {
  // doesn't matter
} else {
  return x / 16;
}
```

In the then-branch, the path condition $x < 0$ would be available, and in the else-branch, the path condition $x \geq 0$ would be available when dividing x by 16. This fact provides an alternative justification for the optimization we discussed above, where a signed division by 16 of a non-negative integer can be turned into a right-shift by four bit positions.

In the *normalization* step of peephole generalization, our goal is to turn every precondition into a dataflow precondition. Dataflow preconditions are a sweet spot because they concisely and abstractly capture facts about values in a program, they can be computed efficiently, and because LLVM already supports them and uses them heavily to drive its peephole optimizations. On the other hand, preconditions that are established by concrete instructions are undesirable because there are typically many different syntactical forms that this kind of precondition can take, and it would be impractical and inefficient to try to pattern match all of them. Path conditions are undesirable because they are too expressive; a compiler like LLVM has no machinery for matching arbitrary predicates, and it is not clear that such a mechanism could be added without slowing down the compiler unacceptably.

Turning implicit preconditions into dataflow preconditions. We first identify a subexpression that is present on both the LHS and the RHS of the optimization. If a subexpression is present on both sides, then none of its instructions are going to be rewritten by the optimization, making it a candidate for replacement by dataflow facts. We visit these eligible subexpressions in order of decreasing size. For each candidate subexpression, we replace it—on both the LHS and RHS—with a fresh free variable, and then attempt to synthesize a dataflow precondition that makes the new optimization valid. The synthesis procedure that we use here is designed to be fast and reliable: we synthesize a *singleton abstract value* whose concretization set contains only a single element. Thus, these fresh dataflow preconditions are undesirably strong—weakening them is a separate step (Section 5.4). If synthesis fails, we move on to a different subexpression. When we run out of eligible subexpressions, any remaining instructions (that cannot be turned into dataflow preconditions) are left in place.

Turning path conditions into dataflow preconditions. We first remove trivial path conditions—those that can be removed without breaking the optimization. The remaining path conditions are required for the optimization to be valid, and can be turned into dataflow preconditions using the same procedure that we use when turning instructions into dataflow preconditions. That is, for each non-trivial path condition that we remove, we attempt to replace it with a maximally strong dataflow condition that we will subsequently weaken. If this synthesis step fails for any given path condition, generalization fails. When this sort of failure occurs, it is generally because the path

condition expressed a fact that the non-relational dataflow analyses that we support are too weak to capture. Fixing this issue—for example by adding relational abstract domains to LLVM—is not within the scope of our work.

5.4 Generalizing Dataflow Preconditions

Dataflow preconditions, regardless of whether they come from normalization or from the original, ungeneralized optimization, should be as weak as possible, while still justifying the optimization.

Weakening concrete dataflow preconditions. For known bits and demanded bits, we incrementally weaken the dataflow fact, one bit at a time, until no weakening is possible without breaking the optimization. Since these domains are structured as *separable* collections of independent bits, this process requires at most n solver invocations for an n-bit SSA value. Moreover, since these abstract domains are lattices, we are guaranteed to arrive at the globally unique weakest precondition.

Our procedure for weakening integer range preconditions is to use binary search to look for the widest possible range that results in a valid optimization. Alas, the integer range abstract domain that LLVM supports is not based on a lattice [Gange et al. 2013]; this stems from the way that it is designed to model both signed and unsigned wraparound using a single representation. Therefore, we are not guaranteed to find the weakest precondition for integer ranges, but rather only a locally weakest precondition.

Generalizing dataflow preconditions with dataflow variables. Since concrete dataflow preconditions—as we have seen—can block generalization, we attempt to replace them with dataflow variables. The process for doing this has the same structure as the process for turning concrete constants into symbolic constants, from Section 4. First, we replace the concrete dataflow precondition with a fresh unconstrained dataflow variable of the appropriate type. If this change breaks the optimization, we attempt to constrain the dataflow variable with one or more predicates. We find these by the same method described in Section 4.3.

Consider the following ungeneralized optimization:

$$\text{Integer-Range}(x) \sqsubseteq [7..42) \models x <_u 7479 \Rightarrow \text{true}$$

This optimization is valid because all the possible values of x are less than 7479, and hence the expression $x <_u 7479$ is always true. We generalize this optimization by replacing the concrete constant (7479) with a fresh symbolic constant C and adding a new predicate $C >_u x.uMax$. The generalized form of this optimization is:

$$C >_u x.\text{uMax} \models x <_u C \Rightarrow \text{true}$$

6 WIDTH INDEPENDENCE

Some compilers internally support a fixed set of types, such as 32-bit and 64-bit integers. LLVM, however, internally supports—and encourages the use of—integer values of any width. Although LLVM's middle-end optimizers need to cope with integer values of arbitrary size, they are free to assume that all code is well-typed. Correspondingly, they must ensure that optimized code is still well-typed. (We briefly described the width-related aspects of LLVM's internal type system in Section 3.2.)

A *width-independent* peephole optimization is one that is correct regardless of the widths of the SSA values on the LHS. For example, $x + (0 - y) \Rightarrow x - y$ is width-independent. On the other hand, $x \times y \Rightarrow x \,\&\, y$ is only valid when x and y are one-bit integers.[3]

[3]Perhaps surprisingly, this optimization is performed by LLVM. https://gcc.godbolt.org/z/T4YsEnoo9

Some optimizations are *fundamentally* width-dependent: no width-independent version exists. For example, the width dependency of $x \times y \Rightarrow x \,\&\, y$ emerges from the truth tables for the multiplication and bitwise-and operators. The width dependency of an optimization involving LLVM's `bswap` intrinsic, that changes the endianness of a bitvector whose width is a multiple of 16, comes from LLVM itself. Our goal with respect to fundamentally width-dependent optimizations is simply to handle them properly by protecting them with appropriate width checks in their preconditions.

Other optimizations are *incidentally* width-dependent: a width-independent form exists and our goal is to find it. The most common source of incidental width dependency is the presence of literal constants in optimizations, regardless of whether they are in the LHS, RHS, or in a precondition. Fortunately, the process of turning literal constants into symbolic constants—which is something that we need to do anyhow, because it is an important aspect of generalization—is often sufficient to create a width-independent optimization.

6.1 An Indirect Approach to Synthesizing Bitwidth-Independent Optimizations

The output of our generalization pipeline up to this point is a peephole optimization that has been proved correct at a single bitwidth. There is no guarantee that this optimization will work at any other width. However, as we previously observed, the part of generalization where we replace literal constants with symbolic constants goes a long way towards exposing the naturally width-independent version of an optimization (if this exists).

Beyond relying on symbolic constants to make optimizations width-independent, we found it necessary to make operand bitwidths available as synthesis components. Consider this ungeneralized optimization:

$$(x : \mathrm{i32} \ll 10) \ll 14 \Rightarrow x \ll 24$$

It could be generalized to:

$$(C_1 + C_2 < 32) \models (x : \mathrm{i32} \ll C_1) \ll C_2 \Rightarrow x \ll (C_1 + C_2)$$

The precondition is required because, in LLVM, shifting a value past its bitwidth results in a poison value (a form of undefined behavior). However, this precondition is problematic because it makes the optimization bitwidth-dependent. A better generalization would be:

$$(C_1 + C_2 < W_x) \models (x \ll C_1) \ll C_2 \Rightarrow x \ll (C_1 + C_2)$$

where W_x is a symbolic width constant that evaluates to the bitwidth of x. This optimization is width-independent.

The next problem is to ensure that the width-independent version gets synthesized. Our solution—based on the insight that width-independent optimizations almost never contain non-trivial literal constants—is to bias synthesis away from width-dependent expressions by blocking the synthesis of literal constants other than zero and one (at this level the highly useful constant -1 is expressed as $\mathrm{sext}(1)$: sign-extending a 1-bit value containing a one to whatever width is needed). It is evident that the constant *one* can be used to construct other constants, defeating the purpose of this restriction. We prevent that by disallowing sub-expressions that fold to other concrete constants—for example we would prevent $1 + 1$ from being synthesized, but allow $W_x - 1$.

In summary, we developed an *indirect* approach to width-independence based on making sure that synthesis has access to a symbolic width variable, and then also biasing the search away from parts of the space that we know are often incidentally width-dependent. This often finds the desired result, but this is not guaranteed. Of course, a direct approach—that synthesizes width-independent optimizations from first principles—would be preferable. We leave this for future research.

6.2 Verifying Width Independence

We take the same brute-force approach to dealing with varying bitwidths that Alive [Lopes et al. 2015] and Alive2 [Lopes et al. 2021] have taken: instantiate and verify an optimization at all feasible widths, up to a configurable maximum width, which we currently set to 64 bits. We do this by erasing the concrete integer types in the optimization and then asking Alive2 to enumerate all width assignments that respect LLVM's type rules.

If an optimization is valid for all feasible widths of its inputs and its output, then it is width-independent, and no width checking is required before applying the optimization. However, when an optimization is valid for a subset of widths, we want synthesize a precondition over widths in one of the following forms:

- a range of widths, e.g.: $1 \leq W \leq 8$
- a set of widths, e.g.: $W \in \{2, 4, 8\}$
- simple inequalities, e.g.: $W_1 \leq W_2$
- power of two, e.g.: $W = 2^n$
- a single width, e.g.: $W = 32$

So far we have not seen optimizations that require width preconditions to be more expressive than this.

As expected from a brute-force approach, verifying width-independence can be very slow for some optimizations. Consider the expression

$$\text{zero-extend}(\text{truncate}(x))$$

The width of x, the width of the truncated value, and the width of the zero-extended value are all constrained only loosely, and there are 124,992 feasible width assignments, considering bitvectors of 64 bits or fewer. To cope with optimizations that have multiple width-changing instructions, we support an optional shortcut mode that only checks widths that are powers of two. This provides, in some cases, multiple orders of magnitude speedup; it does not sacrifice soundness since we encode the power-of-2 restriction on widths into the preconditions of optimizations generated in width-shortcut mode. In the longer term, we would like to avoid brute force and instead invoke a decision procedure that directly supports bitwidth-independent bitvector reasoning; this is an active field of research [Ekici et al. 2023; Niemetz et al. 2019].

7 MAKING OPTIMIZATIONS EXECUTABLE

One route to deployment for a generalized optimization is for a compiler developer to manually translate it into the compiler's implementation language. Although Hydra's internal IR is not particularly readable, we have implemented a pretty-printer whose output closely resembles the format that we use for describing optimizations in this paper.

Another route to deployment, that we explore in this section, is to automate this process by turning generalized optimizations into code. Mechanically-generated code has the advantages of requiring less effort to create, and (presumably) being less error-prone than hand-written code.

Hydra's internal IR supports significantly richer preconditions than does LLVM, meaning that not every optimization in Hydra's IR can be automatically converted into C++ code suitable for inclusion in LLVM. For example, Hydra can easily express a non-linear predicate such as $a = b * c$, but LLVM has no mechanism for checking this (when a, b, and c are arbitrary SSA values). In our experience so far, about a quarter of optimizations generalized from Souper's output are too expressive to be included in LLVM. We hope to eventually encounter a compiler that lets us use these more expressive preconditions.

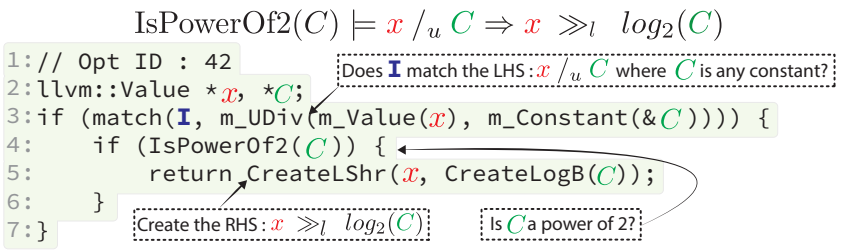

Fig. 6. Example code generated from a peephole optimization

Generating code. Given an optimization, we need to convert it into C++ that inspects and transforms instructions using LLVM's internal APIs and also a small library of auxiliary code that we wrote. Figure 6 shows an optimization that converts unsigned division by a power of two into a right shift, and also its corresponding C++ code (cleaned up slightly, for legibility).

The `match` call in Line 3 invokes a pattern-matching LLVM subsystem that is heavily used by its own peephole optimizations. This call returns true if the instruction `I` is an unsigned division, where the divisor is a literal constant. If so, as a side effect, the dividend is stored in `x` and the divisor is stored in `C`. Although it is not the case here, when trying to match a commutative operation in the left hand side, we emit an LLVM matching primitive that matches the operands in either order.

Line 4 checks the optimization's precondition. Finally, Line 5 creates the optimization's RHS and returns it. All uses of the original SSA value that was produced by the division instruction will be replaced by this new value, and then a subsequent dead code elimination pass will remove the divide and any other instructions that have been rendered dead.

An optimization pass. We wrote an optimization pass template that roughly follows the high-level design of LLVM's InstCombine implementation. Given an LLVM function, it adds each instruction into a queue, and then it processes the queue until empty. As each instruction is removed from the queue, the pass attempts to apply each optimization to it; when an optimization fires, it is replaced with a new instruction, and then each instruction that is an SSA use of the new instruction is appended to the queue. When the queue is empty, a fixpoint has been reached and the pass terminates.

8 EVALUATION

In this section we look at the following research questions:

(1) Are Hydra's generalized optimizations sound?
(2) Do they match more often than their ungeneralized counterparts do?
(3) Can a Hydra-generated pass be competitive with a hand-written peephole optimizer?
(4) Can Hydra effectively automate the construction of executable optimizations given a pair of snippets of LLVM IR that exemplify an ungeneralized optimization?

We answer these questions and then Section 8.7 shows a few additional examples of generalized optimizations. All experiments in this section were run using a version of Hydra that works with LLVM 17.

8.1 Creating an Optimization Pass using Hydra

Sections 8.2–8.4 look at different aspects of an optimization pass that we created using Hydra's output. To create this pass, we first used the Souper superoptimizer to synthesize peephole optimizations for the C and C++ programs in the SPEC CPU 2017 integer benchmark suite, running

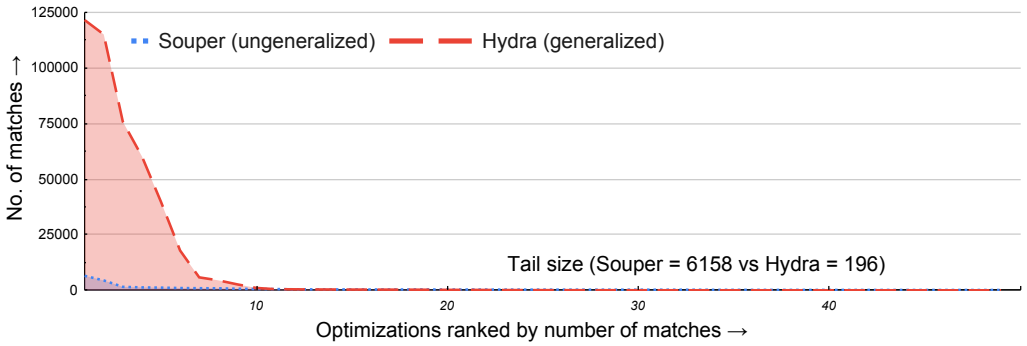

Fig. 7. Generalization works: generalized optimizations match more often, when compiling the integer benchmarks from SPEC CPU 2017

in a mode where InstCombine (LLVM's primary peephole optimizer) was disabled. We directed Souper to synthesize at most one new instruction per RHS, resulting in a set of 26,162 distinct optimizations. Second, we fed these ungeneralized optimizations to Hydra, which—on a 32-core Intel machine—generalized at least one aspect of 77% of them in about seven hours. Each Hydra invocation ran single-threaded and was given 10 minutes of total CPU time before we declared a timeout and killed it. After generalization, there were 7861 distinct optimizations. We discarded those that have preconditions that are too expressive for LLVM to check (as described in Section 7) and then greedily selected a subset of the remaining optimizations that do not subsume each other. This final subset contained 196 optimizations that we converted these optimizations into C++.

8.2 Evaluating End-to-End Correctness

Hydra is a formal-methods-based tool: at almost every step of the way it invokes Z3, via Alive2, to ensure that the optimizations it manipulates are sound. Even so, we want to ensure that Hydra does not produce unsound optimizations, for example by misusing an internal Alive2 API or by mis-translating optimizations from its internal IR to C++.

We compiled the C and C++ benchmarks in the SPEC 2017 integer benchmark suite using a version of Clang/LLVM with InstCombine disabled, and with the Hydra pass described in Section 8.1 running at every point where InstCombine would have run. Hydra's optimizations fired 450,493 times while compiling these benchmarks. We then ran the SPEC benchmarks, using the SPEC driver scripts to confirm that they produced the expected results for the inputs in all three of its configurations (test, train, and ref).

8.3 Evaluating Whether Generalized Optimizations Fire More Often

Next, we again compiled the integer benchmarks from SPEC CPU 2017 using an LLVM with InstCombine disabled, and calling the Hydra-generated pass at every point where InstCombine would have been run. The Hydra-generated pass consists of generalized versions of the optimizations found by Souper, as described in Section 8.1. We also compiled the same benchmarks using the optimizations generated by Souper. Figure 7 shows that generalized optimizations fire much more frequently than do the ungeneralized optimizations discovered by Souper. The top 10 generalized optimizations cumulatively match 23.7x more times than the top 10 ungeneralized optimizations.

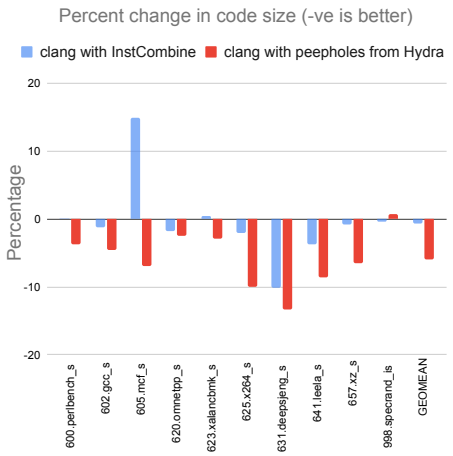

Fig. 8. Code size change for SPEC CPU 2017

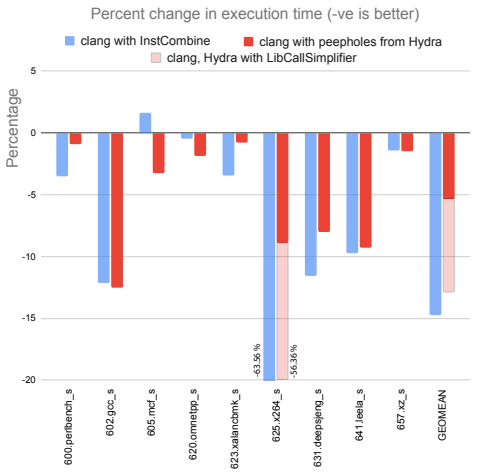

Fig. 9. Execution time change for SPEC CPU 2017

8.4 Case Study: Comparison with InstCombine

This section makes a direct comparison between InstCombine and our Hydra-generated LLVM pass (Section 8.1), in terms of their ability to optimize the integer parts of SPEC CPU 2017. InstCombine sets a high bar: it is one of the most extensive collections of peephole optimizations that we have ever seen. Since InstCombine was split out into its own directory in the LLVM source tree in 2010, it has received more than 5000 commits by over 450 LLVM developers.

We compiled the SPEC CPU 2017 integer benchmarks in three ways:

- using LLVM/Clang 17 at its `-O2` optimization level
- using LLVM/Clang 17 at its `-O2` optimization level, but with the InstCombine pass disabled; this configuration is our baseline
- using LLVM/Clang 17 at its `-O2` optimization level, with InstCombine disabled, and with our Hydra-generated pass running at every point in the optimization pass pipeline where InstCombine would have run

We ran the resulting executables on a single core of an otherwise idle 32-core x86-64 machine (AMD Ryzen 2990WX) with 128 GB of RAM, running Ubuntu 20.04.

Effectiveness of generalized optimizations. Figures 8 and 9 respectively show the change in code size, and the change in execution time, of the benchmark programs, due to the two compiler configurations that we are evaluating, with respect to the baseline LLVM configuration. Lower (more negative) numbers are better in both cases. Figure 9 does not contain 999.specrand because its execution is not timed by the SPEC benchmark driver.[4]

Figure 8 shows that, broadly speaking, the Hydra-generated pass appears to be comparable to, or better than, InstCombine in terms of being able to reduce code size. Notice the significant (about 15%) code size regression for InstCombine on 605.mcf_s. This happens because, for this benchmark, InstCombine happens to reduce the code size of several functions to the point that LLVM's inlining heuristic decides to inline them, which increases the code size by about 5 KB. Hydra does not trigger this behavior, but this was a matter of luck.

[4] https://www.spec.org/cpu2017/Docs/benchmarks/999.specrand_ir.html

Figure 9 shows that, again broadly speaking, the Hydra pass appears to be comparable in power to InstCombine. 625.x264 is an exception: the performance due to Hydra falls far short of the performance when optimized by InstCombine. We looked into this and found that a large part of InstCombine's speedup is due to just one optimization: replacing a call to the `abs()` function in libc with an inlined absolute value implementation. Specializing library calls is not within the current scope of our work. However, to verify that this was the only major factor holding Hydra back, we manually added a call from the Hydra pass to LLVM's library call simplifier. The resulting benchmark performance, shown in a lighter red color in Figure 9, is comparable with InstCombine's.

Effect on compile time. Our Hydra pass slows down the Clang/LLVM compiler by about 4%, compared to its native `-O2` optimization level. The matching code generated by Hydra is simply not as efficient as is the hand-written code in InstCombine, which has been tweaked for speed over a number of years by LLVM engineers, and which uses techniques such as sharing common sub-matching problems across a family of related optimizations. Hydra is a research prototype and we have not yet tuned its output for speed.

Note that unlike InstCombine, a Hydra-generated pass can be easily tuned for speed, simply by selecting fewer optimizations, or it can be tuned to maximum effectiveness by selecting many or all generalized optimizations. We have empirically found that—even in its current, unoptimized state—we can create an optimizer that slows down `clang -O2` by less than 0.5% by selecting only the top 100 optimizations. We leave a full evaluation of the tunability of automatically generated peephole optimizers for future work.

8.5 Evaluating Hydra's Ability to Generalize an Old Superoptimizer's Output

A talk at the LLVM Developers' Meeting 13 years ago described an unsound superoptimizer that worked by extracting fragments from optimized LLVM IR, enumerating potential optimizations, and testing them stochastically [Sands 2011]. It describes (on slide seven) the goals of the work as being:

(1) Automatically find simplifications missed by the LLVM optimizers
(2) And have a human implement them in LLVM

Hydra, of course, automates step (2). In the slides for this talk, we identified 17 optimizations; we encoded each of them as a pair of small LLVM functions.[5]

We generalized the optimizations implied by these pairs of functions; this succeeded in 14 cases, timed out in two, and failed in one case. Alas, most of the generalizations weren't particularly interesting because the optimizations lacked interesting constants. For example, one of them was:

$$x : \mathrm{i32} - (x + y) \Rightarrow 0 - y$$

In cases like this, Hydra proved the optimization correct (recall that this superoptimizer was unsound), proved that it was bitwidth-independent, and generated a C++ implementation suitable for inclusion in LLVM. These things are valuable but do not really exercise Hydra to a significant extent. We do not know if the relative simplicity of these optimizations is an indication of the immaturity of LLVM in 2011, or whether they stem from limitations of this early superoptimizer.

The most interesting generalization from this batch of optimizations was turning this rewrite:

$$x : \mathrm{i32} = ((y + x) + \mathrm{0xFFFFFFFF}) \Rightarrow y = 1$$

into:

[5]These optimizations can be found at https://alive2.llvm.org/ce/z/JDFVxV

$$x = (C + (y + x)) \Rightarrow y = (0 - C)$$

8.6 Evaluating Hydra's Ability to Generalize Contemporary LLVM Issues

In September 2023 we ran a GitHub search for all issues in the official LLVM repository tagged as both `missed-optimization` and `llvm:instcombine`. This yielded 98 issues, 41 open and 57 closed. Of these issues, 40 mentioned desirable optimizations that we judged to be appropriate for our evaluation: they required at least one generalization besides proving width-independence, and were within the scope of Hydra—they involved integer arithmetic instructions. These issues were filed between 2019 and 2023: a period of time when LLVM's peephole optimizers were already quite mature. Thus, they are not low-hanging fruit: many of them are interesting and difficult.

Happily, many of these LLVM issues contained a link to the Compiler Explorer instance that makes Alive2 available on the web. When possible, we used the unoptimized+optimized code found there as the ungeneralized optimization. For most of the rest of the issues, we used Souper to synthesize a valid optimization. In two cases, we had to manually identify and write down the ungeneralized optimization. Some of the issues present a small family of peephole optimizations instead of just one; in these cases we simply picked one of them, with an eye towards choosing the one that seemed to be the defining problem, when this was possible.

Hydra is able to fully generalize 30 out of the 40 optimizations (75%). For the ones Hydra could not generalize, we were able to use the width reduction technique described in Section 6 to find a reduced-width version of the optimizations. This might be useful for compiler developers to manually reason about the optimizations, given smaller constant values. Table 2 summarizes the results of this evaluation, with links to the relevant issues.

We categorize Hydra's failures into three categories:

(1) *Synthesis Failure*: Hydra is unable to generalize the optimization due to a limitation in its synthesis engine. Typically this is because of complex preconditions involving more than two conjunctions.
(2) *Width Independence Failure*: Hydra is unable to prove that the generalized optimization is width independent, usually because the underlying SMT solver times out.
(3) *Timeout*: Hydra fails to generalize the optimization within 10 minutes.

8.7 Optimization Examples

Here we describe a few optimizations from the peephole pass from Section 8.1.

Converting unsigned division by a literal constant that is a power of two into a logical right-shift is an absolutely canonical peephole that most optimizing compilers would support:

$$\text{IsPowerOf2}(C) \models x /_u C \Rightarrow x \gg_u \log_2(C)$$

This related optimization converts an unsigned remainder operation into a bitwise and:

$$\text{IsPowerOf2}(C) \models x \,\%_u\, C \Rightarrow x \,\&\, (C - 1)$$

The ungeneralized version of this optimization used 64-bit values and was difficult for Z3 to reason about; once Hydra reduced the bitwidth to eight, queries became about fifty times faster.

This optimization eliminates a conditional move instruction:

$$(C_1 - C_2) = 1 \models (x \,\&\, 1) \neq 0 \,?\, C_1 : C_2 \Rightarrow C_2 + (x \,\&\, 1)$$

These bitwidth-independent optimizations involve values of different widths:

$$\text{trunc}(C \mid \text{zext}(x)) \Rightarrow x \mid \text{trunc}(C)$$

$$\text{sext}(x \,?\, C_1 : C_2) \Rightarrow x \,?\, \text{sext}(C_1) : \text{sext}(C_2)$$

Table 2. Qualitative evaluation of Hydra's ability to generalize desirable peephole optimizations mentioned in LLVM's issue tracker; these issues were filed from 2019–2023. We discarded the ones that were already manually generalized, or if the only remaining aspect of generalization was width independence. W_x denotes the width of x; sext and zext are the sign-extend and zero-extend operations, with sext(1) being a bitwidth-independent way to express the all-ones value. sMin and sMax are bitwidth-independent versions of the smallest and largest two's complement integer values. cttz is the "count trailing zeroes" operator, ctlz is "count leading zeroes," and ctpop is the Hamming weight operator. The "nsw," "nuw," and "exact" qualifiers on arithmetic instructions are related to undefined behavior [LLVM-LangRef 2023]

Issue	Original	Generalized
66733	$((x:\text{i32} \gg_u 15) \oplus -1)$ & zext$((x = 0)) \Rightarrow$ zext$(x = 0)$	$\sim(x \gg C)$ & zext$(x = 0) \Rightarrow x = 0$
65968	$C = \text{0x80000000} \models (((x:\text{i32}\ \&\ C) \oplus C) =$ $(((y:\text{i32} \oplus -1)\ \&\ C) \oplus C) \Rightarrow (y \oplus x) <_s 0$	$((x\ \&\ \text{sMin}) \oplus \text{sMin}) =$ $((\sim y\ \&\ \text{sMin}) \oplus \text{sMin}) \Rightarrow (y \oplus x) <_s 0$
64305	$((x:\text{i32}\ \%_s\ 8) <_s 0)\ ?\ ((x\ \%_s\ 8) +_{\text{nsw}} 8)\ :\ (x\ \%_s\ 8)$ $\Rightarrow x\ \&\ 7$	IsPowerOf2$(C) \models (x\ \%_s\ C) <_s 0\ ?\ C+$ $(x\ \%_s\ C) : x\ \%_s\ C$ $\Rightarrow x\ \&\ (C-1)$
63472	$C:\text{i1}\ ?\ 64\ :\ 1 + \text{sext}((C \oplus 1)) \Rightarrow C\ ?\ 64\ :\ 0$	$(x\ ?\ C_1 : C_2) + \text{sext}(\sim x) \Rightarrow x\ ?\ C_1 : (C_2 - 1)$
62701	$4 <_u (4 <_u x:\text{i64})\ ?\ (x + -4)\ :\ x \Rightarrow$ $8 <_u x$	Synthesis Failure
62263	$((x:\text{i32}\ \mid\ (y:\text{i32} \oplus -1)) = 0)\ ?\ x\ :\ 0 \Rightarrow 0$	$(x \mid y) = 0\ ?\ x : 0 \Rightarrow 0$
62163	$(x:\text{i32} \gg_s 1) /_s (1\ \%_s\ x) \Rightarrow x \gg_s 1$	$x /_s (1\ \%_s\ x) \Rightarrow x$
62155	$((x:\text{i32}\ \&\ \text{0x7FFFFFFF}) = 0)\ ?\ 0\ :\ (x \ll 1) \Rightarrow x \ll 1$	$(x\ \&\ \text{sMax}) = 0\ ?\ 0 : x \ll 1 \Rightarrow x \ll 1$
61650	zext$((\text{trunc}(a:\text{i32}) \ll 8)) \Rightarrow$ $(a \ll 8)\ \&\ \text{0xFF00}$	zext$(\text{trunc}(x) \ll C) \Rightarrow$ $(x \ll \text{zext}(C))\ \&\ \text{zext}(\sim C)$
61644	$1 \ll ((\text{ctpop}(x:\text{i32}) = 1)\ ?\ 31\ :\ 32 - \text{ctlz}(x))$ $\Rightarrow 1 \ll ((0 - \text{ctlz}((x-1)))\ \&\ 31)$	Width Independence Failure
61538	$a:\text{i8} \times_{\text{nsw}} (((b:\text{i8} \times_{\text{nsw}} (a \ll_{\text{nsw}} 1)) \mid 1) <_s 1)\ ?$ $b\ :\ 1 \Rightarrow ((b \times_{\text{nsw}} a) <_s 0)\ ?$ $(b \times_{\text{nsw}} a)\ :\ a$	$a \times (((b \times_{\text{nsw}} (a \ll_{\text{nsw}} 1)) \mid 1) <_s 1\ ?\ b : 1) \Rightarrow$ $(b \times a) <_s 0\ ?\ (b \times a) : a$
61223	$(x:\text{i32}\ \&\ 1) \neq 0 \models (x + (-1)) \gg_u 1 \Rightarrow$ $x \gg_u 1$	$C_1 \geq 0 \wedge C_1 = -C_2 \models (x + C_1) \gg_u C_2 \Rightarrow$ $x \gg_u C_2$
61183	$(x:\text{i32} = 0)\ ?\ 0\ :\ (1 \ll (32 - \text{ctlz}((x \gg_u 1)))) \Rightarrow$ $(x = 0)\ ?\ 0\ :\ (\text{0x80000000} \gg_u \text{ctlz}(x))$	Width Independence Failure
60801	$(x:\text{i32} = 0)\ ?\ 0\ :\ (x\ \&\ ((1 \ll_{\text{nuw}} \text{cttz}(x))$ $\oplus\ -1)) \Rightarrow x\ \&\ (x + (-1))$	$x = 0\ ?\ 0 : (x\ \&\ \sim(1 \ll \text{cttz}(x)))$ $\Rightarrow x\ \&\ (x-1)$
60799	$(x:\text{i32} = 0)\ ?\ 0\ :\ (1 \ll_{\text{nuw}} \text{cttz}(x)) \Rightarrow x\ \&\ (0 - x)$	$x = 0\ ?\ 0 : 1 \ll \text{cttz}(x) \Rightarrow x\ \&\ (0 - x)$
60242	$(x:\text{i32} \gg_u 31) = \text{zext}((-1 <_s y:\text{i32})) \Rightarrow$ $(x <_u \text{0x80000000}) \oplus (-1 <_s y)$	$C = (W_x - 1) \models (x \gg_u C) = \text{zext}(z:\text{i1}) \Rightarrow$ $z \oplus (x <_u ((\text{sext}(1) \gg_u 1) \oplus \text{sext}(1)))$
60173	$x:\text{i32} + (((x+1)\ \&\ 3) = 0)\ ?\ -2\ :\ 1 \Rightarrow$ $(((x+1)\ \&\ 3) = 0)\ ?\ (x + (-2))\ :\ (x+1)$	$x + (C_4 = (C_2\ \&\ (C_3 + x))\ ?\ C_1 : C_3) \Rightarrow$ $C_4 = (C_2\ \&\ (C_3 + x))\ ?\ : C_1 + x : C_3 + x$
59680	$((x:\text{i32}\ \&\ \text{0xFFFF0000}) = \text{0x11220000})$ $\mid ((x\ \&\ \text{0xFFFFFF00}) = \text{0x11223300})$ $\Rightarrow (x\ \&\ \text{0xFFFF0000}) = \text{0x11220000}$	$C_4 <_u C_3 \models (C_1 = (C_2\ \&\ x)) \mid (C_3 = (C_4\ \&\ x))$ $\Rightarrow C_1 = (C_2\ \&\ x)$
59519	zext$((x:\text{i3} \oplus 7)) \mid \text{sext}(x) \Rightarrow \text{sext}(x) \mid 7$	zext$((\sim x)) \mid \text{sext}(x) \Rightarrow$ zext$(\text{sext}(1)\ \text{to}\ W_x) \mid \text{sext}(x)$
59482	$(\text{zext}(x:\text{i8}) +_{\text{nsw,nuw}} \text{zext}((\text{0xFF} -_{\text{nuw}} x)))$ $<_u \text{0x100} \Rightarrow 1$	Timeout
59266	$((c:\text{i1}\ \&\ (x:\text{i32} = 15)) \oplus (c \mid (x \neq 0))) \oplus 1$ $\Rightarrow x = c\ ?\ 15\ :\ 0$	$\sim((y\ \&\ (x = C_1)) \oplus (y \mid (x \neq C_2)))$ $\Rightarrow x = y\ ?\ C_1 : C_2$
58523	$64 - ((\text{0x47} -_{\text{nuw}} x:\text{i64})\ \&\ -8) \Rightarrow x\ \&\ \text{0x78}$	Timeout
58342	$(x:\text{i8} -_{\text{u,saturating}} 2) = 5 \Rightarrow x = 7$	$C_1 > 0 \wedge C_2 > 0 \models (x -_{\text{u,saturating}} C_1) = C_2 \Rightarrow$ $x = (C_1 + C_2)$
58313	$(x:\text{i1}\ ?\ (y:\text{i1} \oplus 1)\ :\ 0) = 0 \models (x \oplus 1)\ ?\ y\ :\ 0 \Rightarrow y \oplus x$	$x\ ?\ \sim y : C = C \models \sim x\ ?\ y : C \Rightarrow y \oplus x$
58137	$(y:\text{i32} \times_{\text{nsw}} (x:\text{i32} \times_{\text{nsw}} 2)) /_{\text{s,exact}} (x \times_{\text{nsw}} 2) \Rightarrow y$	$(y \times_{\text{nsw}} z) /_s z \Rightarrow y$

Continued on next page

Table 2 – continued from previous page

Issue	Original	Generalized
57666	$(x{:}\mathrm{i1}\ ?\ 4\ :\ 1\ \&\ y{:}\mathrm{i1}\ ?\ 4\ :\ 1) = 0 \Rightarrow y \oplus x$	$C_1 \neq 0 \wedge C_2 \neq 0 \wedge (C_1\ \&\ C_2) = 0 \models$ $(x{:}\mathrm{i1}\ ?\ C_1\ :\ C_2\ \&\ y{:}\mathrm{i1}\ ?\ C_1\ :\ C_2) = 0 \Rightarrow y \oplus x$
57635	$x{:}\mathrm{i8} \neq 0 \models (x + \mathrm{0xFF}) <_u 7 \Rightarrow x \leq_u 7$	$x \neq 0 \models (x + \mathrm{sext}(1)) <_u C \Rightarrow x \leq_u C$
57576	$(y{:}\mathrm{i128} + (x{:}\mathrm{i128} \times (-1) \ll_u 64)) \ll 64 \Rightarrow y \ll 64$	Width Independence Failure
57542	$(42\ /_s\ x{:}\mathrm{i8}) = 0 \Rightarrow (x + \mathrm{0xD5}) <_u \mathrm{0xAB}$	$C \geq 0 \models (C\ /_s\ x) = 0 \Rightarrow (x + {\sim}C) <_u {\sim}(C + C)$
57532	$x{:}\mathrm{i32} <_s (x \oplus (-1)) \Rightarrow x <_s 0$	$x <_s {\sim}x \Rightarrow x <_s 0$
57531	$x{:}\mathrm{i32} + (x \mid (0 - x)) \Rightarrow x\ \&\ (x + (-1))$	$x + (x \mid (0 - x)) \Rightarrow x\ \&\ (x + \mathrm{sext}(1))$
57381	$(0 -_{\mathrm{nsw}} \mathrm{zext}((x{:}\mathrm{i16} \gg_u 15))) \&$ $(\mathrm{0x3E8} -_{\mathrm{nsw}} \mathrm{sext}(x)) \Rightarrow$ $(x <_s 0)\ ?\ (\mathrm{0x3E8} -_{\mathrm{nsw}} \mathrm{sext}(x))\ :\ 0$	$C_2 = (W_x - 1) \models (0 - \mathrm{zext}((x \gg_u C_2))) \&$ $(C_1 - \mathrm{sext}(x)) \Rightarrow$ $x <_s 1 \ll (C_2 + 1)\ ?\ (C_1 - \mathrm{sext}(x)) : 0$
56562	$(v, o) = x{:}\mathrm{i64}\ \mathrm{umul.with.overflow}\ 5 \models$ $(\mathrm{0x07FFFFFFFFFFFFFF} <_u v) \mid o\ ?\ -1 : v$	Timeout
56124	$\mathrm{zext}((((2 \ll c{:}\mathrm{i32})\ \&\ 14) \neq 0)) \Rightarrow \mathrm{zext}((c <_u 3))$	Synthesis Failure
55739	$((2 \ll x{:}\mathrm{i32})\ \&\ 16) \neq 0 \Rightarrow x = 3$	$\mathrm{IsPowerOf2}(C_1, C_2, C_3) \wedge C_2 = (C_3 \times C_1) \models$ $(C_2\ \&\ (C_3 \ll x)) \neq 0 \Rightarrow x = \log_2(C_1)$
55016	$-2 \gg_{\mathrm{s,exact}} (x{:}\mathrm{i32} + -1) \Rightarrow -3 \gg_s x$	Synthesis Failure
54890	$2 \ll_{\mathrm{nsw}} (x{:}\mathrm{i32} - 1) \Rightarrow 1 \ll_{\mathrm{nsw}} x$	Synthesis Failure
54856	$((x{:}\mathrm{i32}\ \&\ 15) \neq 15)\ \&\ (x <_u 16) \Rightarrow x <_u 15$	$\mathrm{IsPowerOf2}(C_1) \wedge (C_1 - C_2) = 1 \models$ $(C_2 \neq (C_2\ \&\ x))\ \&\ (x <_u C1) \Rightarrow x <_u C_2$
51125	$(y{:}\mathrm{i8} \times_{\mathrm{nsw}} ((x{:}\mathrm{i8}\ /_s\ y) +_{\mathrm{nsw}} 1)) -_{\mathrm{nsw}} x$ $\Rightarrow y - (x\ \%_s\ y)$	$(y \times_{\mathrm{nsw}} ((x\ /_s\ y) + 1)) -_{\mathrm{nsw}} x \Rightarrow y - (x\ \%_s\ y)$
41801	$(x{:}\mathrm{i32}\ \&\ (-1 \ll (32 - y{:}\mathrm{i32})))$ $\gg_s (32 - y) \Rightarrow x \gg_s (32 - y)$	$x\ \&\ (\mathrm{sext}(1) \ll y) \gg_s y \Rightarrow x \gg_s y$

$$\mathrm{trunc}(C_3 \mid (x\ ?\ C_1 : C_2)) \Rightarrow x\ ?\ \mathrm{trunc}(C_3 \mid C_1) : \mathrm{trunc}(C_3 \mid C_2)$$

Finally, this rewrite eliminates a bitwise-and instruction when it provably does not matter, because every bit that it clears was not demanded:

$$\mathrm{DemandedBits} = (C\ \&\ \mathrm{DemandedBits}) \models x\ \&\ C \Rightarrow x$$

9 RELATED WORK

Hydra builds on the numerous research results from the last several decades that chip away at difficult engineering problems that are encountered by compiler developers. In this section, we survey this work going roughly from most directly related to Hydra, to least.

9.1 Generalizing Peephole Optimizations

We only know of a few papers that have tackled generalization of peephole optimizations as a first-class concern. Perhaps most closely related to Hydra is Optgen [Buchwald 2015], which could be viewed as combining a superoptimzer with a generalization tool. This integration might be a strength (the entire space of integer peephole optimizations is covered) or a weakness (Optgen does not appear to allow users to generalize a specific, ungeneralized optimization). Optgen works by enumerating all possible preconditions and all possible constant expressions in order of increasing cost. This approach is not guaranteed to find the weakest precondition; our approach of brute-force model counting at reduced bitwidth is better. Optgen exploits the fact that symbolic constants are superior to concrete ones because they are potentially bitwidth-independent, but it only supports optimizations at 8 and 32 bits, while Hydra ensures that all widths up to 64 work. The Optgen paper recognizes that dataflow facts would be useful in peephole optimizations, but does not implement them; it also describes something very much like our dataflow variables, but these also remain unimplemented. Overall, Optgen appears to only be capable of handling much simpler

optimizations than those we have shown that we can handle, e.g. in Table 2. The Optgen work is both practical and innovative, and it was one of the direct inspirations for Hydra.

Proofgen [Tate et al. 2010] takes a proof-centric approach to generalizing optimizations. This approach is very different from ours; it gets a proof from the Peggy infrastructure [Tate et al. 2009], whereas our proofs remain implicit; neither Alive2 nor Z3 represents proofs in an explicit form. This work is, in principle, extremely powerful, but it has not yet been shown to scale to the kinds of generalizations that Hydra can perform. The Proofgen paper contains a number of very interesting, but unevaluated, ideas such as combining a superoptimizer with a generalization engine.

9.2 Synthesis and Peephole Optimizations

Peephole optimizations have existed for at least 60 years [McKeeman 1965], and automatically generating peephole optimizations is, consequently, a well researched topic. PO [Davidson and Fraser 1984] is one of the earliest efforts to generate peephole optimizations automatically. This approach involved simulating all possible pairs and triples of machine instructions, abstracting the register transfers, and searching for a cheaper instruction that produced the same result for a set of tests. As an interesting bit of trivia, GCC and LLVM have inherited the nomenclature of calling a peephole optimizer a *combiner* from PO.

A superoptimizer takes a collection of instructions and searches for a better (or, the best) set of instructions that has the same effect. This relies either on testing or on solver-based verification. The first superoptimizer [Massalin 1987] tested all possible alternative rewrites to find a potentially correct one, which had to be checked by a human for correctness. More recent superoptimizers [Bansal and Aiken 2006; Joshi et al. 2002; Sasnauskas et al. 2017; Schkufza et al. 2013] utilize SMT solvers to provide a guarantee that the optimizations discovered are sound. A solver such as Z3 [De Moura and Bjørner 2008] or CVC4 [Barrett et al. 2011] is typically used to solve queries in the theory of quantified bitvectors.

There is a considerable amount of work on using synthesis to optimize bitvector programs. A large fraction of these projects are based on the SyGuS [Alur et al. 2013] initiative, which provides a common language platform for synthesis tools. This includes a divide and conquer approach [Alur et al. 2017], a probabilistic approach [Lee et al. 2018], and one based on unification [Alur et al. 2015]. There are two meta-approaches for making program synthesis faster: pruning and guiding. With pruning, the aim is to reduce an exponentially large search space to a size small enough for enumeration. LENS [Phothilimthana et al. 2016], MCSYNTH [Srinivasan et al. 2016], and Souper [Mukherjee et al. 2020] have distinct approaches for pruning the bit-vector synthesis search space. The other approach, guiding, is about finding a good result early in the search. Type guided synthesis [Guo et al. 2019] is an approach which scales for program synthesis in the presence of polymorphic types and functions in modern programming languages. Sketching [Bornholt et al. 2016; Solar-Lezama 2008] and component-based synthesis [Shi et al. 2019; Tiwari et al. 2015] are two other approaches to find good results faster. Hydra uses sketching by adapting subexpressions from the left hand side to synthesize constant expressions on the right hand side. We also utilize a limited form of component-based synthesis by adding some common bit-vector operations as enumeration components.

9.3 Program Transformations Beyond Compilers

Program transformations are useful in many domains beyond compilers. There is a large body of work on program transformation in the context of refactoring [Negara et al. 2013] and program repair [Goues et al. 2019]. These use cases for program transformation are different from compilers in the sense that they are not trying to optimize the program, but change it for achieving goals like improving readability, filling in missing pieces of code, or fixing bugs.

Table 3. A brief summary of how four contemporary production compilers express peephole optimizations, and how they would implement $x \times 1 \Rightarrow x$

LLVM: 40,000 lines of C++ with an embedded pattern-matching DSL. Example:

```
if (match(I, m_c_Mul(
  m_Value(x), m_SpecificInt(1))))
    return x;
```

GCC: 8,200 lines of match.pd, compiled to C++. Example:

```
(simplify
  (mult:cs @0 integer_onep)
  @0)
```

Go: 17,400 lines of DSL, compiled to Go. Example:

```
(Mul(8|16|32|64)
  (Const(8|16|32|64)[1]) x)
  => x
```

Cranelift: 2735 lines of ISLE, compiled to Rust. Example:

```
(rule (simplify (imul ty
  x (iconst ty (u64_from_imm64 1))))
(subsume x))
```

Program transformations—independent of the domain—are often required to be general, i.e., they should work for a wide range of programs instead of just a specific program. SYDIT [Meng et al. 2011], REFAZER [Rolim et al. 2017], Genesis [Long et al. 2017], and GenPat [Jiang et al. 2020] are tools that generate patches to transform programs. These tools have a notion of generalization that is meant to ensure that the patches are applicable beyond the training set. SYDIT abstracts away program locations and concrete identifiers to generalize the edit scripts. REFAZER and Genesis deal with multiple input-output examples to identify the common transformation that is inherent in the examples, thus producing a more general patch. GenPat starts from a single example and generalizes it by learning from a large corpus of code. The hypothesis is that keeping the constructs present more frequently in the corpus are likely to lead to a more general patch.

Hydra's preprocessing step (Section 3) is a simpler form of the generalization step in these tools. We can get away with this simpler, greedy technique because we have a specific domain where smaller LHSs guarantee a more general transformation, and we have a fast verification phase that makes it easy to try a large number of possibilities.

9.4 Finding Weakest Preconditions

Generalized peephole optimizations have symbolic constants and expressions involving (potentially constrained) symbolic constants, as opposed to the concrete constants found in ungeneralized peepholes. PSyCO [Lopes and Monteiro 2014] treats this problem as a weakest precondition inference problem. Alive-Infer [Menendez and Nagarakatte 2017] extends Alive to provide a data-driven approach to solve the precondition inference problem. Given several positive and negative input examples, Alive-Infer utilizes a greedy set cover algorithm to find a predicate that rejects all the negative examples and accepts as many positive examples as possible. This approach is similar to PIE [Padhi et al. 2016], which is a tool for inferring general purpose weakest preconditions from a set of test executions. Finding weakest preconditions is a well studied problem with a rich history; here we describe only a subset that is closely related to compiler optimizations. Hydra, necessarily, has to compromise on this aspect of generalization to make the problem tractable, as we have to try and find a precondition for a large number of candidate constant expressions.

9.5 Peephole Optimizers in Contemporary Production Compilers

We looked in detail at how four open-source compilers—LLVM, GCC, the Go compiler, and Cranelift—express peephole optimizations. Table 3 describes the major peephole optimizers found in these

compilers, and shows how they would express a simple optimization. LLVM's primary SSA-level peephole optimizer is InstCombine—short for Instruction Combiner; it is implemented in C++, making heavy use of pattern-matching primitives that effectively function as a simple embedded DSL. The `m_c_Mul` call is a commutative matching primitive. Hydra generates code in this same embedded DSL (Section 7). GCC has two intermediate representations on which peephole optimizations are performed: the higher level GIMPLE and the lower level RTL. Table 3 shows a rewrite in the match.pd[6] DSL for optimizations on GIMPLE. GCC also has a significant number of peephole optimizations written in C++. The Go compiler uses a DSL to express both middle-end and back-end peephole optimizations; this optimizer is smaller in scope compared to GCC's and LLVM's, and is geared towards fast compilation. Finally, Cranelift is a fast compiler that is used to translate WebAssembly into native code, and is also gaining traction as an alternative backend for the Rust compiler. It has a collection of peephole optimization rules, implemented in the ISLE DSL. The majority of the rules in the Cranelift compiler are in its architecture specific backends. In all four cases, the high-level goal is to enable compiler developers to produce readable, modifiable, general purpose optimizations that are suitable for inclusion in production compilers. Ironically, of these four compilers, LLVM has the best tool for formally verifying peephole optimizations [Lopes et al. 2021], but the worst implementation language for them. Regardless, we believe that techniques like ours are complementary to modern peephole optimization implementation techniques, and that tools like Hydra should be able to produce these rules automatically, given specific examples produced by humans or by superoptimizers.

10 FUTURE WORK

While we have demonstrated a viable prototype for peephole optimization generalization, there remain several avenues for future research.

Broadening the scope. LLVM's optimization infrastructure automatically applies our generalized scalar optimizations to vector instructions in element-wise fashion. However, we cannot yet generalize operations that explicitly use vectors, such as permutations. Similarly, we have not yet supported loads and stores, nor instructions that process floating point values. We are not aware of conceptual difficulties in generalizing peephole optimizations that make use of these program features, but more research and development work is needed to make this happen. Another interesting direction is to generalize optimizations that work for multiple compilers. Although we hope that others will be successful in adopting and improving the techniques described in this paper, we believe that it should be possible to support multiple compilers using a single toolchain.

Richer preconditions. A tantalizing possibility is synthesizing transfer functions [Kalita et al. 2022] for interesting abstract domains, plugging these into a generic fixpoint iteration, and then synthesizing peephole optimizations that make use of the resulting dataflow facts. This method would seem to offer a relatively cheap way of discovering whether a compiler can make use of, for example, arithmetic congruences [Granger 1989; King and Søndergaard 2010].

11 CONCLUSION

Compiler developers, and also compiler users, routinely identify situations where the compiler's output is suboptimal, but could be fixed by giving the compiler a local rewrite rule—a peephole optimization—that removes the inefficiency. However, in practice, implementing a sound and appropriately general rewrite rule is a highly skilled task, and even experts miss corner cases, leading to compiler bugs. Our thesis is that compiler implementers would be well supported by

[6] https://medium.com/@prathamesh1615/adding-peephole-optimization-to-gcc-89c329dd27b3

formal-methods-based automation for optimization generalization. We designed and implemented Hydra, a tool that fills this gap using program synthesis. Among other benefits, Hydra provides a direct path for superoptimization to be useful in improving a production compiler: the peephole superoptimizer finds specific opportunities for improvement and generalization turns these into broadly usable optimizations.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1955688. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

Alessandro Abate, Cristina David, Pascal Kesseli, Daniel Kroening, and Elizabeth Polgreen. 2018. Counterexample guided inductive synthesis modulo theories. In *International Conference on Computer Aided Verification*. Springer, 270–288.

Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. *Formal Methods in Computer-Aided Design*, 1–17. https://doi.org/10.1109/FMCAD.2013.6679385

Rajeev Alur, Pavol Černý, and Arjun Radhakrishna. 2015. Synthesis Through Unification. In *Computer Aided Verification*, Daniel Kroening and Corina S. Păsăreanu (Eds.). 163–179. https://doi.org/10.1007/978-3-319-21668-3_10

Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017. Scaling Enumerative Program Synthesis via Divide and Conquer. In *Tools and Algorithms for the Construction and Analysis of Systems*, Axel Legay and Tiziana Margaria (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 319–336. https://doi.org/10.1007/978-3-662-54577-5_18

Sorav Bansal and Alex Aiken. 2006. Automatic Generation of Peephole Superoptimizers. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA). 394–403. https://doi.org/10.1145/1168918.1168906

Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Computer Aided Verification*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 171–177.

James Bornholt, Emina Torlak, Dan Grossman, and Luis Ceze. 2016. Optimizing Synthesis with Metasketches. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. 775–788. https://doi.org/10.1145/2837614.2837666

Sebastian Buchwald. 2015. Optgen: A Generator for Local Optimizations. In *24th International Conference on Compiler Construction (CC 2015)*, Björn Franke (Ed.). London, UK, 171–189.

José Cambronero, Sumit Gulwani, Vu Le, Daniel Perelman, Arjun Radhakrishna, Clint Simon, and Ashish Tiwari. 2023. FlashFill++: Scaling Programming by Example by Cutting to the Chase. *Proc. ACM Program. Lang.* 7, POPL, Article 33 (jan 2023), 30 pages. https://doi.org/10.1145/3571226

Jack W. Davidson and Christopher W. Fraser. 1984. Automatic Generation of Peephole Optimizations. In *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction* (Montreal, Canada) *(SIGPLAN '84)*. Association for Computing Machinery, New York, NY, USA, 111–116. https://doi.org/10.1145/502874.502885

Leonardo De Moura and Nikolaj Bjørner. 2008. 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* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

Bruno Dutertre. 2015. Solving Exists/Forall Problems With Yices. In *13th International Workshop on Satisfiability Modulo Theories (SMT 2015)*.

Burak Ekici, Arjun Viswanathan, Yoni Zohar, Cesare Tinelli, and Clark Barrett. 2023. Formal Verification of Bit-Vector Invertibility Conditions in Coq. In *Frontiers of Combining Systems*, Uli Sattler and Martin Suda (Eds.). Springer Nature Switzerland, Cham, 41–59.

Graeme Gange, Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter James Stuckey. 2013. Abstract Interpretation over Non-lattice Abstract Domains. In *SAS*.

Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62, 12 (nov 2019), 56–65. https://doi.org/10.1145/3318162

Philippe Granger. 1989. Static analysis of arithmetical congruences. *International Journal of Computer Mathematics* 30, 3–4 (1989), 165–190. https://doi.org/10.1080/00207168908803778 arXiv:https://doi.org/10.1080/00207168908803778

Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *POPL '11*.

Zheng Guo, Michael James, David Justo, Jiaxiao Zhou, Ziteng Wang, Ranjit Jhala, and Nadia Polikarpova. 2019. Program Synthesis by Type-Guided Abstraction Refinement. *Proc. ACM Program. Lang.* 4, POPL, Article 12 (Dec. 2019). https://doi.org/10.1145/3371080

Susmit Jha and Sanjit A. Seshia. 2017. A theory of formal synthesis via inductive learning. *Acta Informatica* 54, 7 (2017), 693–726. https://doi.org/10.1007/s00236-017-0294-5

Jiajun Jiang, Luyao Ren, Yingfei Xiong, and Lingming Zhang. 2020. Inferring program transformations from singular examples via big code. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering* (San Diego, California) *(ASE '19)*. IEEE Press, 255–266. https://doi.org/10.1109/ASE.2019.00033

Rajeev Joshi, Greg Nelson, and Keith Randall. 2002. Denali: a goal-directed superoptimizer. In *Proceedings of the ACM SIGPLAN 2002 conference on Programming language design and implementation* (Berlin, Germany) *(PLDI '02)*. 304–314.

Pankaj Kumar Kalita, Sujit Kumar Muduli, Loris D'Antoni, Thomas Reps, and Subhajit Roy. 2022. Synthesizing Abstract Transformers. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 171 (oct 2022), 29 pages. https://doi.org/10.1145/3563334

Andy King and Harald Søndergaard. 2010. Automatic Abstraction for Congruences. In *Verification, Model Checking, and Abstract Interpretation*, Gilles Barthe and Manuel Hermenegildo (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 197–213.

Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. 2018. Accelerating Search-based Program Synthesis Using Learned Probabilistic Models. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. 436–449. https://doi.org/10.1145/3192366.3192410

LLVM-LangRef. 2023. LLVM Language Reference Manual. http://llvm.org/docs/LangRef.html. Accessed: 9-1-2023.

Fan Long, Peter Amidon, and Martin Rinard. 2017. Automatic inference of code transforms for patch generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 727–739. https://doi.org/10.1145/3106237.3106253

Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: Bounded Translation Validation for LLVM. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 65–79. https://doi.org/10.1145/3453483.3454030

Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. 2015. Provably Correct Peephole Optimizations with Alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA). 22–32.

Nuno P. Lopes and José Monteiro. 2014. Weakest Precondition Synthesis for Compiler Optimizations. In *Verification, Model Checking, and Abstract Interpretation*, Kenneth L. McMillan and Xavier Rival (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 203–221.

Henry Massalin. 1987. Superoptimizer: A Look at the Smallest Program. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems* (Palo Alto, California, USA) *(ASPLOS '87)*. 122–126. https://doi.org/10.1145/36177.36194

W. M. McKeeman. 1965. Peephole Optimization. *Commun. ACM* 8, 7 (July 1965), 443–444.

David Menendez and Santosh Nagarakatte. 2017. Alive-Infer: Data-Driven Precondition Inference for Peephole Optimizations in LLVM. *SIGPLAN Not.* 52, 6 (jun 2017), 49–63. https://doi.org/10.1145/3140587.3062372

Na Meng, Miryung Kim, and Kathryn S. McKinley. 2011. Systematic editing: generating program transformations from an example. *SIGPLAN Not.* 46, 6 (jun 2011), 329–342. https://doi.org/10.1145/1993316.1993537

Manasij Mukherjee, Pranav Kant, Zhengyang Liu, and John Regehr. 2020. Dataflow-Based Pruning for Speeding up Superoptimization. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 177 (nov 2020), 24 pages. https://doi.org/10.1145/3428245

Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph E. Johnson, and Danny Dig. 2013. A Comparative Study of Manual and Automated Refactorings. In *ECOOP 2013 – Object-Oriented Programming*, Giuseppe Castagna (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 552–576.

Aina Niemetz, Mathias Preiner, Andrew Reynolds, Yoni Zohar, Clark Barrett, and Cesare Tinelli. 2019. Towards Bit-Width-Independent Proofs in SMT Solvers. In *Automated Deduction – CADE 27*, Pascal Fontaine (Ed.). Springer International Publishing, Cham, 366–384.

Saswat Padhi, Rahul Sharma, and Todd Millstein. 2016. Data-Driven Precondition Inference with Learned Features. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 42–56. https://doi.org/10.1145/2908080.2908099

Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. 2016. Scaling Up Superoptimization. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) *(ASPLOS '16)*. 297–310. https://doi.org/10.1145/2872362.2872387

Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning syntactic program transformations from examples. In *Proceedings of the 39th International*

Conference on Software Engineering (Buenos Aires, Argentina) *(ICSE '17)*. IEEE Press, 404–415. https://doi.org/10.1109/ICSE.2017.44

Duncan Sands. 2011. Super-optimizing LLVM IR. http://llvm.org/devmtg/2011-11/Sands_Super-optimizingLLVMIR.pdf Presentation at the 2011 LLVM Developers' Meeting.

Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Gratian Lup, Jubi Taneja, and John Regehr. 2017. Souper: A Synthesizing Superoptimizer. arXiv:1711.04422 [cs.PL]

Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic Superoptimization. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Houston, Texas, USA) *(ASPLOS '13)*. 305–316. https://doi.org/10.1145/2451116.2451150

Kensen Shi, Jacob Steinhardt, and Percy Liang. 2019. FrAngel: Component-based Synthesis with Control Structures. *Proc. ACM Program. Lang.* 3, POPL, Article 73 (Jan. 2019), 73:1–73:29 pages. https://doi.org/10.1145/3290386

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

Venkatesh Srinivasan, Tushar Sharma, and Thomas Reps. 2016. Speeding Up Machine-code Synthesis. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands) *(OOPSLA 2016)*. 165–180. https://doi.org/10.1145/2983990.2984006

Jubi Taneja, Zhengyang Liu, and John Regehr. 2020. Testing static analyses for precision and soundness. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization* (San Diego, CA, USA) *(CGO 2020)*. Association for Computing Machinery, New York, NY, USA, 81–93. https://doi.org/10.1145/3368826.3377927

Ross Tate, Michael Stepp, and Sorin Lerner. 2010. Generating Compiler Optimizations from Proofs. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Madrid, Spain) *(POPL '10)*. Association for Computing Machinery, New York, NY, USA, 389–402. https://doi.org/10.1145/1706299.1706345

Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality Saturation: A New Approach to Optimization. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Savannah, GA, USA) *(POPL '09)*. Association for Computing Machinery, New York, NY, USA, 264–276. https://doi.org/10.1145/1480881.1480915

Ashish Tiwari, Adrià Gascón, and Bruno Dutertre. 2015. Program Synthesis Using Dual Interpretation. In *Automated Deduction - CADE-25*, Amy P. Felty and Aart Middeldorp (Eds.). Springer International Publishing, Cham, 482–497.

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. 283–294. https://doi.org/10.1145/1993316.1993532

Received 21-OCT-2023; accepted 2024-02-24