Weakest Precondition Synthesis for Compiler Optimizations

Nuno P. Lopes and José Monteiro

INESC-ID, IST Universidade de Lisboa

Abstract. Compiler optimizations play an increasingly important role in code generation. This is especially true with the advent of resource-limited mobile devices. We rely on compiler optimizations to improve performance, reduce code size, and reduce power consumption of our programs.
Despite being a mature field, compiler optimizations are still designed and implemented by hand, and usually without providing any guarantee of correctness.
In addition to devising the code transformations, designers and implementers have to come up with an analysis that determines in which cases the optimization can be safely applied. In other words, the optimization designer has to specify a precondition that ensures that the optimization is semantics-preserving. However, devising preconditions for optimizations by hand is a non-trivial task. It is easy to specify a precondition that, although correct, is too restrictive, and therefore misses some optimization opportunities.
In this paper, we propose, to the best of our knowledge, the first algorithm for the automatic synthesis of preconditions for compiler optimizations. The synthesized preconditions are provably correct by construction, and they are guaranteed to be the weakest in the precondition language that we consider.
We implemented the proposed technique in a tool named PSyCO. We present examples of preconditions synthesized by PSyCO, as well as the results of running PSyCO on a set of optimizations.

1 Introduction

Compiler optimizations are increasingly important. We rely on them to improve performance, reduce code size, and reduce power consumption of our programs. The advent of mobile devices with limited resources and the need to reduce the operating costs of data centers puts even more pressure on the quality of the results of compiler optimizations.

This demand for improving code efficiency is driving the development of new and more complex optimizations. However, neither the specification nor the implementation of these optimizations is usually proved correct.

In fact, a recent study found bugs in all the most used compilers [39]. These bugs range from mere crashes to subtle wrong-code emission.

Ensuring that compilers are correct is of extreme importance. All the programs we produce, in one way or another, are processed by compilers. If the compilers are not proved correct, properties formally verified at the source-code level of a program are not carried to the binary code, since the compiler may introduce bugs during the translation process.

Despite such a strong necessity for compilation quality, compilers are still largely written by hand. Moreover, the source-code of each of the major compilers has several million lines of code. Testing a whole code base is therefore unlikely to be practical to accomplish.

This paper gives a step towards improving the situation. We present, to the best of our knowledge, the first algorithm for the automatic synthesis of the *weakest* precondition for compiler optimizations specified in a high-level language. These preconditions are provably correct by construction.

We consider a language of preconditions for compiler optimizations consisting of read and write sets for template statements (and expressions), which we believe to be adequate to express the most used conditions in this domain. Given a compiler optimization specified in a high-level template language, our algorithm synthesizes the *weakest* precondition in terms of read and write sets (assuming it is solely expressible in terms of these sets).

The generated preconditions can then be either used by the compiler developer to implement an analysis that guarantees that the precondition holds before performing the code transformation, or it can be used by a separate tool to automatically generate such an analysis.

The algorithm works in a counterexample-driven way. It requires a black box that can prove the correctness of a compiler optimization, or produce a counterexample otherwise. Then, the algorithm processes the counterexample and produces the weakest precondition guaranteed to prevent that counterexample. The algorithm repeats this process until no counterexamples are possible, i.e., when the optimization is correct (which is guaranteed to occur, since the set of possible preconditions is finite).

Our algorithm is more generally applicable than the domain of compiler optimizations. The algorithm can synthesize weakest preconditions for any problem where the set of preconditions is finite, although possibly too large to test each case individually, as long as there exists an oracle that can prove correctness or produce counterexamples if not correct.

The rest of the paper is organized as follows. Section 2 gives a set of preliminary definitions. Section 3 gives an intuition of how our algorithm synthesizes weakest preconditions for compiler optimizations through a simple example. Section 4 describes our algorithm for the synthesis of weakest preconditions for compiler optimizations. Section 5 presents PSyCO, a tool that implements the proposed algorithm, as well as examples of preconditions generated by PSyCO and results of running it over a set of compiler optimizations. Section 6 presents the related work.

$$
\begin{aligned}
e &::= n \mid v \mid e_1 \oplus e_2 \mid \mathsf{E}_i \\
b &::= e \leq 0 \mid b_1 \otimes b_2 \mid \neg b_1 \mid \mathsf{B}_i \\
c &::= \textbf{skip} \mid v \ := \ e \mid c_1 \ ; \ c_2 \mid \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2 \mid \textbf{while } b \textbf{ do } c_1 \mid \mathsf{S}_i
\end{aligned}
$$

Fig. 1. Template program syntax. n is an integer number; v is a variable name; E_i are integer template expressions (side-effect free); B_i are boolean template expressions (side-effect free); S_i are template statements; $\oplus$ is a binary operator over integer expressions (e.g., $+$, $-$); and $\otimes$ is a binary operator over boolean expressions (e.g., $\wedge$, $\vee$).

2 Preliminaries

Compiler optimizations are represented by a triple (τ, ψ, h). Transformation function $\tau \equiv Src \Rightarrow Tgt$ is a function that takes an instantiation of the source template program Src and returns the target template program Tgt properly instantiated. An instantiation of a template program is a mapping from all the template statements and expressions to concrete (without templates) statements/expressions. The precondition ψ is a sufficient condition that makes τ semantics-preserving. Finally, h is a profitability heuristic, which states under which conditions the compiler should apply τ, since τ may not always be performance improving. We will ignore the profitability heuristic for the rest of this paper because it does not interfere with the correctness of optimizations.

Template Programs Template programs are specified using the syntax shown in Figure 1. In addition to the normal program features, template programs may have template expressions and template statements. Template expressions are side-effect free expressions whose value is unknown. It may be a constant or it may be an arbitrary algebraic expression that depends on several variables. Similarly, template statements are placeholders for arbitrary statements (e.g., variable assignments, function calls, or even loops).

Side-effect free expressions are allowed to read any number of variables and memory locations (including none), but are not allowed to write to them. Moreover, side-effect free expressions may not raise exceptions nor trap. Such erroneous behaviors must be explicitly modeled in the control flow and/or as assignments to control variables.

Transformation functions state how each template statement/expression from the source program is transformed (e.g., moved, duplicated, eliminated) to produce the target program. For example, consider the following transformation function τ_1:

$$
\begin{array}{ll}
\mathsf{S} & \\
v := \mathsf{E} & \Rightarrow \qquad \begin{array}{l} v := \mathsf{E} \\ \mathsf{S} \end{array}
\end{array}
$$

As an example, we apply the transformation function τ_1 to the following program fragment:

$x := 0$
$v := x + 1$

The output of the transformation function is (with the instantiation $\mathsf{S} \mapsto x := 0,\ \mathsf{E} \mapsto x + 1$):

$v := x + 1$
$x := 0$

The transformed program is not equivalent to the original one, since if the initial value of x is not zero, then the programs will yield different values for v. Therefore, a necessary (but not sufficient) precondition to ensure that the transformation function is always semantics-preserving is that S cannot write to a variable that is read by E.

Preconditions of optimizations are specified as read and write sets of the template statements/expressions, which contain the variables that the template statements/expressions *may* read and write, respectively. Since template expressions are side-effect free, their write set is empty, i.e., for every template expressions E_i and B_i, we have $\mathsf{W}(\mathsf{E}_i) = \emptyset$ and $\mathsf{W}(\mathsf{B}_i) = \emptyset$.

For the previous example, we could use the condition $\mathsf{W}(\mathsf{S}) \cap \mathsf{R}(\mathsf{E}) = \emptyset \wedge v \notin \mathsf{R}(\mathsf{S}) \wedge v \notin \mathsf{W}(\mathsf{S})$ as the precondition. This precondition is sufficient to ensure that the transformation function is always semantics-preserving, and it would therefore rule out the instantiation above.

Let $\mathsf{Tmpl}(\tau)$ be the set of template statements/expressions of the transformation function τ. Let $\mathsf{Stmts}(\tau) \subseteq \mathsf{Tmpl}(\tau)$ be the set of template statements of the transformation function τ. In our example, we have $\mathsf{Tmpl}(\tau_1) = \{\mathsf{S}, \mathsf{E}\}$ and $\mathsf{Stmts}(\tau_1) = \{\mathsf{S}\}$.

Program Paths A program path π is a sequence of straight-line statements (no loops nor **if** statements) and boolean expressions. For example, the path $i := 0\ ;\ i < n\ ;\ i := i + 1\ ;\ i \geq n$ could be a 1-step unrolling of a simple counting loop. We naturally extend $\mathsf{Stmts}(\pi)$ and $\mathsf{Tmpl}(\pi)$ to program paths.

Context Variables Let $c_i \in C$ be a context variable. These variables c_i represent the variables that are possibly in scope where a program template may be instantiated (possibly none) and that do not appear in the transformation function.

For the example above we need two context variables and therefore we have $C = \{c_1, c_2\}$. Variable c_1 represents, e.g., the effects of S on x. While variable x does not appear explicitly in the transformation function, S does indeed modify x in the example instantiation. Variable c_2 represents all the other eventual variables of the program that are in scope (possibly none) that can be read by E and that cannot be written by S.

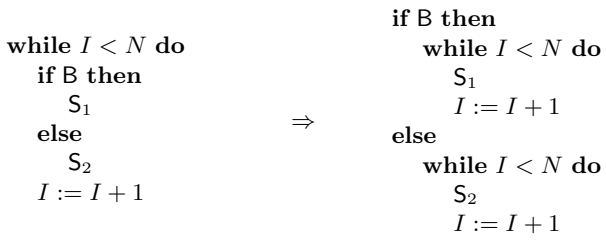

Fig. 2. Loop unswitching: the source template is on the left, and the target template is on the right.

For every template statement S_i, we have that the context variable $c_i = \mathsf{CtxVar}(\mathsf{S}_i)$ is always in its write set, i.e., $c_i \in \mathsf{W}(\mathsf{S}_i)$. Therefore, each template statement may write to at least one distinct context variable.

In order to not restrict the generated preconditions, we require our precondition synthesis algorithm to be run with at least one more context variable than template statements, i.e., for transformation function τ we must have $|C| \geq |\mathsf{Stmts}(\tau)| + 1$. This lower bound on the size of C is sufficient to express all combinations of constraints of the form $\mathsf{R}(t) \cap \mathsf{W}(s_1) = \emptyset$ and $\mathsf{W}(s_1) \cap \mathsf{W}(s_2) = \emptyset$ (and their respective negations) for any template t and statements s_1 and s_2. Constraints of the form $\mathsf{R}(t_1) \cap \mathsf{R}(t_2) = \emptyset$ are not considered, since we are not aware of any optimization requiring such kind of preconditions.

Let $\mathsf{Vars}(\tau)$ and $\mathsf{Vars}(\pi)$ be the set of variables in a transformation function τ or in a path π, respectively. Moreover, context variables are contained in these sets, i.e., $C \subseteq \mathsf{Vars}(\tau)$. In our example, we have $\mathsf{Vars}(\tau_1) = \{v, c_1, c_2\}$.

Miscellaneous Throughout the paper, we use the constraint $x = \mathrm{ite}(a, b, c)$ as the usual shorthand for $a \to x = b \wedge \neg a \to x = c$.

3 Illustrative Example

We illustrate our algorithm to synthesize weakest preconditions for compiler optimizations on a simple example. Figure 2 shows an optimization known as loop unswitching. Intuitively, this optimization looks correct iff B evaluates to the same boolean value in every iteration (i.e., B must be loop invariant).

S_1 and S_2 are template statements. They are placeholders that represent an arbitrary statement, such as a variable assignment, a loop, or a compound statement. Similarly, B is a template (side-effect free) boolean expression. We do not know what these statements and expressions exactly do (this is only defined when the optimization is applied to a specific piece of code), and so we need to derive a generic precondition to restrict their operation to guarantee that the transformation will be always semantics-preserving.

The language of preconditions we use for compiler optimizations is that of read and write sets of template statements/expressions. For example, to state

that the template expression B cannot read variable I we use the notation $I \notin \mathsf{R}(\mathsf{B})$. This condition is actually necessary (but not sufficient) for the precondition of loop unswitching.

Our algorithm works in a counterexample-driven way. We require the existence of a black box that can prove the correctness of compiler optimizations, or produce a counterexample if the optimization is not correct.

For our example, starting with the precondition $P = \mathsf{true}$, we could get the following paths π_1 and π_2 (respectively, for the source and target templates):

$$I < N \ ; \ \mathsf{B} \ ; \ \mathsf{S}_1 \ ; \ I := I + 1 \ ; \ I < N \ ; \ \neg\mathsf{B} \ ; \ \mathsf{S}_2 \ ; \ I := I + 1 \ ; \ I \geq N$$

and:

$$\mathsf{B} \ ; \ I < N \ ; \ \mathsf{S}_1 \ ; \ I := I + 1 \ ; \ I < N \ ; \ \mathsf{S}_1 \ ; \ I := I + 1 \ ; \ I \geq N$$

Without any knowledge about S_1 and S_2, these paths are clearly a counterexample since S_1 and S_2 execute a different number of times in the source and target templates. For example, the instantiation $\mathsf{S}_1 \mapsto I := I+1,\ \mathsf{S}_2 \mapsto I := I + 2,\ \mathsf{B} \mapsto I \leq 0$ makes the paths of the source and target programs terminate with different values for variable I. Therefore, we need to constrain the set of possible instantiations of S_1 and S_2 with a suitable precondition.

To generate a precondition for a given counterexample, we first encode the counterexample paths into logic in the usual way, with the variables of the target template being renamed in order to be different from the variables used in the source template. We explain only how template statements and expressions are encoded. Each template statement is treated as a conditional assignment to all variables of the program by a fresh variable. Then, for each pair of the same template symbol, we assert that their corresponding fresh variables are equal iff the values of their corresponding input variables that are in the read set are equal. Similarly, template expressions are replaced by fresh variables.

For our counterexample, we obtain the following constraint ϕ_1 for the source path (with $\mathsf{Vars}(\pi_1;\pi_2) = \{I, N, c_1, c_2, c_3\}$):

$$
\begin{aligned}
&I_0 < N_0 \wedge \\
&B_0 \wedge \\
&I_1 = \mathrm{ite}(w^I_{S1}, S1^I_0, I_0) \wedge N_1 = \mathrm{ite}(w^N_{S1}, S1^N_0, N_0) \wedge \\
&c1_1 = \mathrm{ite}(w^{c1}_{S1}, S1^{c1}_0, c1_0) \wedge c2_1 = \mathrm{ite}(w^{c2}_{S1}, S1^{c2}_0, c2_0) \wedge c3_1 = \mathrm{ite}(w^{c3}_{S1}, S1^{c3}_0, c3_0) \\
&\wedge I_2 = I_1 + 1 \wedge \\
&I_2 < N_1 \wedge \\
&\neg B_1 \wedge \\
&I_3 = \mathrm{ite}(w^I_{S2}, S2^I_0, I_2) \wedge N_2 = \mathrm{ite}(w^N_{S2}, S2^N_0, N_1) \wedge \\
&c1_2 = \mathrm{ite}(w^{c1}_{S2}, S2^{c1}_0, c1_1) \wedge c2_2 = \mathrm{ite}(w^{c2}_{S2}, S2^{c2}_0, c2_1) \wedge c3_2 = \mathrm{ite}(w^{c3}_{S2}, S2^{c3}_0, c3_1) \\
&\wedge I_4 = I_3 + 1 \wedge \\
&I_4 \geq N_2
\end{aligned}
$$

The encoding (ϕ_2) of the target path is similar.

The boolean variables w_s^v mean that statement s writes to variable v. This is required because $v \in \mathsf{W}(s)$ states that s *may* write to variable v, but it is not mandatory to do so. We define ϕ_w to be the conjunction of the following set of constraints (for each pair of template statement s and variable v):

$$w_s^v \rightarrow v \in \mathsf{W}(s)$$

We now generate the constraints ϕ_u that assert when the fresh values generated from the template statements/expressions are equal. These constraints are akin to Ackermann's reduction for uninterpreted function symbols. For example, to state when B_0 and B_1 are equal, we use the following constraint:

$$\big((I \in \mathsf{R}(\mathsf{B}) \rightarrow I_0 = I_2) \wedge (N \in \mathsf{R}(\mathsf{B}) \rightarrow N_0 = N_1) \wedge (c_1 \in \mathsf{R}(\mathsf{B}) \rightarrow c1_0 = c1_1) \wedge \\ (c_2 \in \mathsf{R}(\mathsf{B}) \rightarrow c2_0 = c2_1) \wedge (c_3 \in \mathsf{R}(\mathsf{B}) \rightarrow c3_0 = c3_1)\big) \rightarrow B_0 = B_1$$

Set membership constraints ($x \in y$) are encoded as boolean variables.

Now that we have all the necessary constraints, we construct the following formula ϕ and give it to an SMT solver:

$$\forall V\ (\phi_1 \wedge \phi_2 \wedge \phi_w \wedge \phi_u \rightarrow \\ I_4 = I_8 \wedge N_2 = N_4 \wedge c1_2 = c1_4 \wedge c2_2 = c2_4 \wedge c3_2 = c3_4)$$

where I_4/I_8, N_2/N_4, $c1_2/c1_4$, $c2_2/c2_4$, and $c3_2/c3_4$ are the final values of the $I/N/c_1/c_2/c_3$ variables of the source and target templates, respectively. V is the set of variables that are universally quantified. These include the variables w_s^v, and all the fresh variables created by the path encoding process.

Giving this formula to an SMT solver will yield assignments to the boolean variables corresponding to the read and write sets' membership (the only existentially quantified variables). Each set of these assignments (a model of the formula) is a sufficient precondition that makes the two paths equivalent (or unreachable).

For this formula, we may obtain the model $\mathsf{R}(\mathsf{B}) = \emptyset \wedge \mathsf{W}(\mathsf{S}_1) = \{c_1\} \wedge \mathsf{W}(\mathsf{S}_2) = \{c_2\}$. Although this condition is certainly sufficient to make the first path unreachable (because it implies that B is a constant, and therefore it cannot evaluate to two different values), this condition is not the weakest.

As we stated before, our algorithm is iterative and so it keeps weakening the counterexample's precondition until it gets the weakest precondition. We do so by negating each model, adding it to formula ϕ, and then retrieving another model from the SMT solver. We stop when there are no more models (i.e., the conjunction of ϕ and the negation of all the previously discovered models is unsatisfiable). The weakest precondition for the counterexample is the disjunction of all models.

After processing one counterexample, we strengthen the transformation function's weakest precondition with the counterexample's weakest precondition. We iterate until there are no more counterexamples, i.e., until the transformation function is correct.

The algorithm terminates because the precondition is strengthened when each counterexample is processed and because the language of preconditions we consider is finite.

Finally, the precondition we obtain for our example (loop unswitching) after processing all the counterexamples is the following:

$$P = I \notin \mathsf{R(B)} \wedge \mathsf{W(S_1)} \cap \mathsf{R(B)} = \emptyset \wedge \mathsf{W(S_2)} \cap \mathsf{R(B)} = \emptyset$$

Optimizations The algorithm we just presented informally will take significant time to terminate, since it will usually enumerate many models. We present two optimizations that improve the speed of convergence significantly, as well as improve the compactness of the generated preconditions.

The first optimization we perform is model weakening, meaning that given a model generated by an SMT solver, we try to make it weaker (more general) by dropping literals from it (which are therefore "don't cares"). For a model μ of ϕ, we know that $\neg\phi \wedge (\bigwedge l \in \mu)$ is unsatisfiable. Moreover, if for some literal l', $\neg\phi \wedge (\bigwedge l \in \mu' = \mu \setminus \{l'\})$ is still unsatisfiable, then we know that both $\mu' \cup \{l'\}$ and $\mu' \cup \{\neg l'\}$ are models of ϕ. Therefore, μ' is a weaker (partial) model of ϕ. We leverage this knowledge to iterate over each of the literals in a model to check which ones can be removed.

The second optimization we perform is to add additional constraints to ϕ that represent common precondition patterns. In particular, we noticed that stating that the intersection of read/write sets must be empty (e.g., $\mathsf{W(S_1)} \cap \mathsf{R(B)} = \emptyset$) is a common pattern. We therefore associate a boolean variable to each of such constraints, and try to bias the weakening of the models (as previously described) towards these variables. This optimization not only produces more compact preconditions, but also reduces the number of models considerably (since it avoids enumerating all the models that correspond to the constraint they succinctly imply).

4 The Algorithm

Our precondition synthesis algorithm, named PSyCO, is counterexample-guided. The algorithm relies on a verification tool as a black box that can prove the correctness of optimizations (such as CORK [25] or PEC [20]) or return a counterexample otherwise.

4.1 PSyCO

The pseudo-code for the PSyCO algorithm is shown in Figure 3. The algorithm takes as input a transformation function τ and returns the corresponding weakest precondition ψ. Starting with the precondition true, the algorithm iteratively calls the CheckTF function that checks whether τ is correct under the given precondition or returns a counterexample otherwise (given as two paths, π_1 and π_2, of the source and target programs, respectively). The CheckTF function is a black box given as input.

```
    function PSyCO
    input
      τ – transformation function
    vars
      ψ – generated precondition
    begin
1     ψ := true
2     repeat
3       match CheckTF(τ, ψ) with
4         | correct ->
5           return ψ
6         | counterexample (π1, π2) ->
7           ψ := ψ ∧ SynthWP(π1, π2)
    end.
```

Fig. 3. PSyCO algorithm.

At each step, PSyCO strengthens the precondition with a condition that is sufficient (and necessary) to discharge the counterexample. Therefore, a given counterexample is never seen more than once. Since the number of possible combinations of preconditions in the considered language is finite and we keep strengthening the precondition at each step, we have that PSyCO terminates (assuming that CheckTF always terminates).

4.2 SynthWP

Figure 4 shows the function SynthWP that takes two paths, π_1 and π_2, as input, and returns the weakest precondition that makes the two paths equivalent.

The idea is to construct an universally quantified formula such that a model for it guarantees that the two paths are equivalent (or either one becomes unreachable) for all possible program inputs. The union of all models is the weakest precondition. The models can be generated with an off-the-shelf SMT solver.

SynthWP starts by generating a formula that corresponds to each of the counterexample paths (lines 3 and 4). This is done using standard techniques, that we do not describe here. VCGen takes as input a path π, a map σ_0 with the initial values of the program variables, a set of variables V containing the variables of the source path, and a map w containing a fresh boolean variable for each pair of statements and variables. If a certain statement writes to a given program variable, its corresponding boolean variable in w will be true.

VCGen replaces each template statement s with the following constraint:

$$\bigwedge_{v \in V} (v' = \text{ite}(w(s, v), fv, v))$$

where v' is the new value of v and fv is a fresh variable (one per variable v). Template expressions are replaced with a fresh variable.

function SynthWP
input
π_1, π_2 – counterexample paths
vars
ψ – generated precondition
begin

1 $\sigma_0 := \{v \mapsto \text{fresh integer var} \mid v \in \mathsf{Vars}(\pi_1; \pi_2)\}$
2 $w := \{(s, v) \mapsto \text{fresh boolean var} \mid s \in \mathsf{Stmts}(\pi_1; \pi_2) \wedge v \in \mathsf{Vars}(\pi_1; \pi_2)\}$
3 $\phi_1, \sigma_1, u_1 := \textsc{VCGen}(\pi_1, \sigma_0, \mathsf{Vars}(\pi_1), w)$
4 $\phi_2, \sigma_2, u_2 := \textsc{VCGen}(\pi_2, \sigma_0, \mathsf{Vars}(\pi_1), w)$
5 $\phi_u := \bigwedge_{(\sigma,v,t),(\sigma',v',t') \in (u_1 \cup u_2) \wedge t = t'} \Big(\Big(\bigwedge_{v'' \in \mathsf{Vars}(\pi_1)} (\mathcal{B}(v'' \in \mathsf{R}(t)) \rightarrow \sigma(v'') = \sigma'(v'')) \Big) \rightarrow v = v' \Big)$
6 $\phi_w := \bigwedge_{((s,v) \mapsto l) \in w} (l \rightarrow \mathcal{B}(v \in \mathsf{W}(s)))$
7 $\phi_c := \bigwedge_{s \in \mathsf{Stmts}(\pi_1; \pi_2)} (\mathcal{B}(\mathsf{CtxVar}(s) \in \mathsf{W}(s)))$
8 $\phi_d, d := \textsc{MkDisj}(\pi_1; \pi_2)$
9 $V := \{\sigma_0(v) \mid v \in \mathsf{Vars}(\pi_1; \pi_2)\} \cup \{l \mid ((s, v) \mapsto l) \in w\} \cup \{v \mid (\sigma, v, t) \in (u_1 \cup u_2)\} \cup d$
10 $\phi := \forall V \ \Big(\phi_u \wedge \phi_w \wedge \phi_c \wedge \phi_d \wedge \phi_1 \wedge \phi_2 \rightarrow \bigwedge_{v \in \mathsf{Vars}(\pi_1)} (\sigma_1(v) = \sigma_2(v)) \Big)$
11 $\mu_f := \{\neg \mathcal{B}(v \in \mathsf{R}(t)) \mid v \in \mathsf{Vars}(\pi_1; \pi_2) \wedge t \in \mathsf{Tmpl}(\pi_1; \pi_2)\} \cup \{\neg \mathcal{B}(v \in \mathsf{W}(s)) \mid v \in \mathsf{Vars}(\pi_1; \pi_2) \wedge s \in \mathsf{Stmts}(\pi_1; \pi_2)\} \cup d$
12 $\psi := \mathsf{false}$
13 **while** $\phi \wedge \neg\psi$ is satisfiable **do**
14 $\psi := \psi \vee \textsc{GeneralizeWP}(\phi, \textsc{GetModel}(\phi) \cap \mu_f, d)$
15 **return** ψ
end.

Fig. 4. SynthWP algorithm.

VCGen returns a formula corresponding to the input path, a map σ with the final value of each of the program variables and a set u. The set u contains triples (σ, v, t), one per each fresh variable v created for template statement/expression t, with σ being a map with the value of the variables at the point where the template statement/expression was evaluated.

In line 5, we generate a formula akin to Ackermann's reduction for uninterpreted function symbols. For each pair of triples (σ, v, t) and (σ', v', t') in w coming from the same template (i.e., $t = t'$), we assert that the fresh variables v and v' must be equal if each of the variables in the read set of t has the same value in σ and σ'. We use the notation $\mathcal{B}(x \in y)$ to introduce a boolean variable that represents that $x \in y$.

In line 6, we generate a constraint that asserts that a template statement can only write to a variable v if v is in its write set. The constraint generated in line 7 asserts that each template statement s_i has at least one distinct context variable c_i in its write set. This is an important optimization, since it avoids the generation of multiple equivalent models that are equal modulo a renaming of the context variables.

In line 8, we introduce a set of boolean variables to represent constraints of the form $\mathsf{W}(s_1) \cap \mathsf{W}(s_2) = \emptyset$ and $\mathsf{R}(t) \cap \mathsf{W}(s_1) = \emptyset$ for every pair of template statements s_1 and s_2 and template statements/expressions t. This is an optimization that enables us to more succinctly express preconditions of this form without having to enumerate all the possible combinations of read and write sets that satisfy the corresponding constraint.

For a path π, MkDisj generates constraints of the form:

$$\bigwedge_{s,s' \in \mathsf{Stmts}(\pi)} \left(\left(\neg \bigvee_{v \in \mathsf{Vars}(\pi)} (\mathcal{B}(v \in \mathsf{W}(s)) \wedge \mathcal{B}(v \in \mathsf{W}(s'))) \right) \leftrightarrow fv \right)$$

with fv being a fresh variable (one per each pair (s, s')). MkDisj generates similar constraints for every pair of read and write sets. In addition to the generated constraint, MkDisj returns the set of fresh variables used.

In line 9, we collect the set of variables that will be universally quantified, namely the set of initial values of the variables and the set of fresh variables used in previous steps. The remaining variables (the booleans representing set membership and the variables in d) are implicitly existentially quantified. Finally, in line 10, we assemble the final constraint. It states that either one of the paths is unreachable or the final value of the variables of the two paths must be equal.

In line 11, we compute a model filter, since we are only interested in negative membership constraints and empty intersection constraints (d). We do not need to consider positive membership constraints, since e.g., $v \in \mathsf{R}(t)$ means that t *may* read v (but not necessarily). Therefore, a model μ including a positive membership constraint l, e.g., $\mu = \mu' \cup \{l\}$, implies that $\mu' \cup \{\neg l\}$ is also a model.

Lines 13–15 implement the main synthesis loop. We iterate over the models of the formula ϕ (filtered by μ_f) and generalize each one in the hope that we will produce more succinct preconditions and converge faster. We pass the set

```
    function GeneralizeWP
    input
      φ – a formula
      μ – a model of formula φ (set of literals)
      ψ – set of preferred literals to bias the solution
    begin
1     if ¬φ ∧ (⋀ l ∈ μ ∩ ψ) is unsatisfiable
2       return MinimizeCore(¬φ, GetUnsatCore(¬φ, μ ∩ ψ))
3     else
4       return MinimizeCore(¬φ, GetUnsatCore(¬φ, μ) ∪ (μ ∩ ψ)))
    end.
```

Fig. 5. GeneralizeWP algorithm.

of variables d to GeneralizeWP, so that it can bias the result and express it over more literals of d whenever possible. Function GetModel is given by the SMT solver and returns a model for the formula given as input.

Encoding Size In the worst-case, the size of formula ϕ is dominated by ϕ_u, since that is the only constraint that grows quadratically with the size of the input. Given a counterexample (π_1, π_2), the worst-case size of ϕ_u (and therefore of ϕ) is $O\left(\left(|\pi_1 \ ; \ \pi_2| \cdot |\mathsf{Vars}(\pi_1)|\right)^2\right)$.

4.3 GeneralizeWP

Figure 5 shows the function GeneralizeWP. As input, it takes a formula ϕ, a model μ of ϕ given as a set of literals, and a set of preferred literals ψ. The purpose of this function is to compute a new model for ϕ, hopefully smaller than μ (and obviously not bigger), while maximizing the set of literals of ψ that will be part of the result.

From the definition of model of a formula, we know that $\neg\phi \wedge (\bigwedge l \in \mu)$ is unsatisfiable. If we a drop a literal, say l', from μ and if $\neg\phi \wedge (\bigwedge l \in \mu \setminus \{l'\})$ is still unsatisfiable, then $\mu' = \mu \setminus \{l'\}$ is a model of ϕ as well. However, μ' contains fewer literals than μ, and is therefore more generic (since we now know that both $\mu' \cup \{l\}$ and $\mu' \cup \{\neg l\}$ are models of ϕ).

Function GeneralizeWP works as follows. First, it checks whether restricting the model to the set of preferred literals is sufficient to make ϕ unsatisfiable. If so, it calls MinimizeCore to further reduce the size of the solution. We use the function GetUnsatCore, which is usually provided by SMT solvers, as an optimization. GetUnsatCore(x, y) returns a set $y' \subseteq y$ such that $x \wedge (\bigwedge l \in y')$ is unsatisfiable.

If the set of preferred literals is not enough, then we call MinimizeCore with the whole model. Since we are not able to bias the result of GetUnsatCore, we need to ensure that the set of preferred literals is passed to MinimizeCore.

```
    function MinimizeCore
    input
      φ – a formula
      ζ – an unsat core of formula φ (set of literals)
    vars
      Ψ – minimized core
    begin
1     Ψ := ∅
2     while ζ ≠ ∅ do
3       κ := take one from ζ
4       if φ ∧ (⋀ l ∈ Ψ ∪ ζ) is satisfiable then
5         Ψ := Ψ ∪ {κ}
6     return Ψ
    end.
```

Fig. 6. MinimizeCore algorithm.

4.4 MinimizeCore

Figure 6 shows the function MinimizeCore. Given a formula ϕ and a set of literals ζ such that $\phi \wedge (\bigwedge l \in \zeta)$ is unsatisfiable, the objective of this function is to find a possibly smaller set $\Psi \subseteq \zeta$ such that $\phi \wedge (\bigwedge l \in \Psi)$ is still unsatisfiable.

MinimizeCore works by checking if each literal $l \in \zeta$ is necessary for the formula to be unsatisfiable. If so, l is added to the result set Ψ. We employ a linear search, as opposed to potentially better search strategies such as QuickXplain [19] or Progression [26], since linear search proved to perform well in our benchmarks.

In our implementation, ζ is a list and we perform a linear search from the beginning to the end of the list. This strategy enables us to bias the search to give priority for removal of certain literals. In particular, in the function GeneralizeWP, we put all the preferred literals ψ at the end of the list, which biases the solution towards having a higher number of literals of ψ.

4.5 Discussion

The proposed algorithm, although agnostic to the verification algorithm used, assumes that only counterexamples for partial functional correctness proofs are generated. This means that the algorithm as presented will produce weakest *liberal* preconditions. To produce weakest preconditions, the algorithm has to be extended so that it can handle counterexamples for relative termination mismatches (based upon, e.g., [5, 8, 17]).

The proposed specification language does not include instructions to access heap locations or arrays. This means that the current algorithm does not handle optimizations that perform explicit transformations to memory access instructions. It does, however, support instantiation of templates with memory accessing instructions (such as instantiating a template expression with a load from a

memory location), provided that the instantiation meets the precondition (which can be verified using, e.g., a data dependency analysis).

In this paper, we only consider preconditions in the language of read and write sets. However, arithmetic preconditions may be needed for some optimizations. For example, a specialization of the loop unrolling optimization requires the number of iterations of the source loop to be even[1]. Synthesizing such preconditions could be done by, for example, adapting the counterexample-driven algorithm of Seghir and Kroening [33].

5 Evaluation

We implemented a prototype named PSyCO[2], which stands for Precondition Synthesizer for Compiler Optimizations. PSyCO is implemented in Python (in about 1,400 lines of code), and uses Z3 4.3.2 [10] for constraint solving.

In principle, PSyCO can be used with any compiler optimization verification tool that can produce counterexamples. However, we chose to implement a simple bounded model checker (BMC) within PSyCO for convenience. This BMC only checks optimizations for partial correctness, and therefore the results we present in this section are weakest *liberal* preconditions. We did not use our own verification tool, CORK [25], since it is several orders of magnitude slower at producing counterexamples than our simple BMC. Furthermore, CORK does not support disjunctive preconditions natively, and therefore it has to first convert such preconditions to DNF and test each of the conjuncts separately.

We show in Table 1 a few examples of compiler optimizations and the corresponding preconditions generated by PSyCO. This list is not supposed to be exhaustive, since there are many optimizations and each one of them may be specified in slightly different ways. We show these examples so that the reader can truly appreciate the simplicity of the generated preconditions and realize how surprising the weakest preconditions can be (from what you would expect at first thought).

There are very few published formally stated preconditions for compiler optimizations. However, the PEC paper [20] does include a precondition for software pipelining (in a slightly different language than the one we used), that was written down by hand and then verified correct by PEC. The precondition synthesized by PSyCO (as shown in Table 1) is, however, weaker than that in PEC's paper. Their precondition requires that $V_1 \notin \mathsf{W}(\mathsf{S}_1)$ and $V_2 \notin \mathsf{W}(\mathsf{S}_1)$, while the precondition generated by PSyCO does not. Therefore, the precondition generated by PSyCO is weaker than that published by experts in formal methods and compiler optimizations, showing that automatic precondition synthesis does indeed help to make optimizations more widely applicable.

[1] We used a more general version of loop unrolling (that accounts for an even and odd number of loop iterations) in our experiments.

[2] Prototype and benchmarks available from `http://web.ist.utl.pt/nuno.lopes/psyco/`.

Optimization	Weakest Liberal Precondition

Partial redundancy elimination (PRE)
if B **then**
  S_1
  $V_1 := \mathsf{E}$
  S_2
else
  S_3
$V_2 := \mathsf{E}$
$\Rightarrow$
if B **then**
  S_1
  $V_1 := \mathsf{E}$
  S_2
  $V_2 := V_1$
else
  S_3
  $V_2 := \mathsf{E}$
	$V_1 \notin \mathsf{R}(\mathsf{E}) \wedge V_1 \notin \mathsf{W}(\mathsf{S}_2) \wedge$
	$\mathsf{R}(\mathsf{E}) \cap \mathsf{W}(\mathsf{S}_2) = \emptyset$

Code hoisting
if B **then**
  S_1
  S_2
else
  S_1
  S_3
$\Rightarrow$
S_1
if B **then**
  S_2
else
  S_3
	$\mathsf{R}(\mathsf{B}) \cap \mathsf{W}(\mathsf{S}_1) = \emptyset$

Loop unrolling
while $V_1 < V_2$ **do**
  S
  $V_1 := V_1 + 1$
$\Rightarrow$
while $(V_1 + 1) < V_2$ **do**
  S
  $V_1 := V_1 + 1$
  S
  $V_1 := V_1 + 1$
if $V_1 < V_2$ **then**
  S
  $V_1 := V_1 + 1$
	$V_2 \notin \mathsf{W}(\mathsf{S}) \wedge$
	$(V_1 \notin \mathsf{W}(\mathsf{S}) \vee$
	$\mathsf{R}(\mathsf{S}) \cap \mathsf{W}(\mathsf{S}) = \emptyset)$

Strength reduction
while $V_1 < V_2$ **do**
  $V_3 := V_1 * \mathsf{E}$
  S
  $V_1 := V_1 + 1$
$\Rightarrow$
$V_4 := V_1 * \mathsf{E}$
while $V_1 < V_2$ **do**
  $V_3 := V_4$
  $V_4 := V_4 + \mathsf{E}$
  S
  $V_1 := V_1 + 1$
	$V_1 \notin \mathsf{R}(\mathsf{E}) \wedge V_3 \notin \mathsf{R}(\mathsf{E}) \wedge$
	$\mathsf{R}(\mathsf{E}) \cap \mathsf{W}(\mathsf{S}) = \emptyset \wedge$
	$\big(V_1 \notin \mathsf{W}(\mathsf{S}) \vee$
	$(V_3 \notin \mathsf{R}(\mathsf{S}) \wedge$
	$\mathsf{R}(\mathsf{S}) \cap \mathsf{W}(\mathsf{S}) = \emptyset)\big)$

Software pipelining
while $V_1 < V_2$ **do**
  S_1
  S_2
  $V_1 := V_1 + 1$
$\Rightarrow$
if $V_1 < V_2$ **then**
  S_1
  while $V_1 < (V_2 - 1)$ **do**
  S_2
  $V_1 := V_1 + 1$
  S_1
  S_2
  $V_1 := V_1 + 1$
	$V_2 \notin \mathsf{W}(\mathsf{S}_2) \wedge$
	$\big((\mathsf{R}(\mathsf{S}_1) \cap \mathsf{W}(\mathsf{S}_2) = \emptyset \wedge$
	$\mathsf{R}(\mathsf{S}_1) \cap \mathsf{W}(\mathsf{S}_1) = \emptyset \wedge$
	$\mathsf{R}(\mathsf{S}_2) \cap \mathsf{W}(\mathsf{S}_2) = \emptyset) \vee$
	$V_1 \notin \mathsf{W}(\mathsf{S}_2)\big)$

Table 1. Examples of weakest preconditions synthesized by PSyCO.

Optimization	# Counterexamples	# Models	WP Time	Total Time
Code hoisting	1	1	0.07s	0.23s
Constant propagation	1	1	0.04s	0.16s
Copy propagation	0	0	0s	0.11s
If-conversion	0	0	0s	0.11s
Partial redundancy elimin.	1	1	0.10s	0.30s
Loop fission	6	36	1.28s	2.18s
Loop flattening	1	1	0.07s	3.31s
Loop fusion	6	36	1.26s	2.19s
Loop interchange	11	25	1.42s	23.8s
Loop invariant code motion	3	3	0.22s	0.55s
Loop peeling	0	0	0s	0.27s
Loop reversal	4	7	0.25s	0.54s
Loop skewing	1	1	0.06s	163s
Loop strength reduction	1	2	1.14s	1.41s
Loop tiling	1	1	0.07s	4.60s
Loop unrolling	2	4	0.13s	0.50s
Loop unswitching	2	2	0.15s	0.77s
Software pipelining	1	2	0.13s	0.58s

Table 2. List of compiler optimizations [1, 28], the number of counterexamples processed, the number of models obtained for preconditions, the time taken by the precondition generation algorithm, and the overall time taken by the tool (including the BMC).

We ran PSyCO over a set of optimizations (mostly loop manipulating). The experiments were run on a machine running Linux 3.10.10 with an Intel Core 2 Duo 3.00 GHz CPU, and 4 GB of RAM. The results are shown in Table 2.

Since we are not aware of any other algorithm for precondition synthesis for compiler optimization, we cannot compare PSyCO against other tools. In Table 2, we show the number of counterexamples required for each optimization to reach convergence. We notice that in general only a few counterexamples are required (with certain optimizations with a trivial precondition requiring none).

Then, we present the total number of models obtained for preconditions. The ratio of the number of models per number of counterexamples is small because of the employed optimizations described before (model weakening and inference of common precondition patterns).

Finally, we show the time taken by the precondition synthesis algorithm, as well as the overall time taken by the tool. The overall time includes not only the synthesis algorithm, but also the BMC time as well as minor initializations performed by the tool. We argue that the time taken by the precondition synthesis algorithm is low. Overall, PSyCO is usually fast, with a few exceptions due to high inefficiencies in the Z3Py module exposed by our BMC. However, the running time for this kind of tool is not critical, since it is supposed to be run off-line, and only once per optimization specification.

6 Related Work

This work is related with both precondition synthesis and compiler (optimizations) correctness. We briefly describe both topics here.

6.1 Precondition Synthesis

The concepts of weakest preconditions (WPs) and weakest liberal preconditions (WLPs) have long been introduced by Dijkstra [11]. Since then, several algorithms have been published to accomplish their automatic generation.

There are several competing approaches for WLP synthesis. These include, for example, precondition templates and constraint solving (e.g., [15]), quantifier elimination (e.g., [27]), abstract interpretation (e.g., [9]), and CEGAR, predicate abstraction, and interpolation for predicate generation (e.g., [33]). Some algorithms combine multiple techniques to achieve better performance.

Our unsat core minimization algorithm, that biases the result towards certain literals, is similar to the one presented by Seghir and Kroening [33].

Leino [21] describes a compact encoding for verification conditions generated from the weakest precondition calculus.

Cook et al. [8] propose a counterexample-driven algorithm for precondition synthesis (not necessarily weakest) to guarantee program termination, and Bozga et al. [5] propose an algorithm based on abstract interpretation.

Calcagno et al. [7] present an algorithm for WLP synthesis based on separation logic.

Gulwani et al. [14] present an algorithm to synthesize loop-free programs that implement a given specification. While the goal of the algorithm is not to synthesize preconditions, there is a similarity in the encoding of program equivalence and in the usage of an SMT solver to find assignments to variables that represent the synthesized artifact.

6.2 Compiler Correctness

Several approaches have been proposed to improve the correctness of compilers, including manual and computer-assisted proofs, automatic verification, and automatic generation of correct optimizations by construction.

CompCert [24] is a compiler that aims to provide end-to-end correctness guarantees (from a program's source code down to the resulting binary). CompCert was written from scratch with verification in mind, and its correctness proofs are done in Coq. Vellvm [41] is a Coq-based framework that enables the development and verification of compiler optimizations for LLVM.

CORK [25] is a compiler optimization verifier based on recurrence computation. PEC [20, 37] (a successor of Cobalt [22] and Rhodium [23]) is also a verifier, but uses bisimulation relation synthesis as the underlying technique. Both CORK and PEC require a precondition to be given as input.

Translation validation (e.g., [13,30,31,34,38,40,42]) is a technique for establishing the correctness of compiler optimizations *after* the optimization was run

by checking the original and optimized programs for equivalence. Namjoshi and Zuck [29] propose augmenting transformation functions so that they generate auxilarly invariants to help the translation validation process, which otherwise could fail to derive those invariants automatically.

Guo and Palsberg [16] present a bisimulation-based technique to reason about the correctness of trace optimizations.

Godlin and Strichman [12] propose a set of proof rules to prove equivalence of programs and to prove mutual termination using uninterpreted function symbols to abstract recursive function calls. The technique is later extended with the introduction of mutual summaries [17].

Relational Hoare logic [4] is an extension to Hoare logic to prove equivalence of programs. Barthe et al. [3] extend this work to support non-structurally equivalent programs.

Superoptimization (e.g., [2, 6, 18, 35]) is a technique to do code optimization given a set of theorems that establish equalities between code sequences and then searching for a better equivalent program.

Tate et al. [36] propose an algorithm to extrapolate compiler optimizations directly from concrete examples.

Scherpelz et al. [32] propose an algorithm to automatically synthesize flow functions from compiler optimizations' preconditions.

7 Conclusion

In this paper we presented, to the best of our knowledge, the first algorithm for the automatic synthesis of weakest preconditions for compiler optimizations. The algorithm generates preconditions iteratively, in a counterexample-driven approach. Preconditions for counterexamples are generated by an SMT solver.

We built a prototype, named PSyCO, that implements the proposed algorithm and we presented several preconditions generated by it.

Acknowledgments. The authors thank the anonymous reviewers for their comments and suggestions on earlier drafts of this paper.

This work was partially supported by the FCT grants SFRH/BD/63609/2009 and INESC-ID multiannual funding PEst-OE/EEI/LA0021/2013.

References

1. A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley, 2006.
2. S. Bansal and A. Aiken. Automatic generation of peephole superoptimizers. In *ASPLOS*, 2006.
3. G. Barthe, J. M. Crespo, and C. Kunz. Relational verification using product programs. In *FM*, 2011.
4. N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *POPL*, 2004.

5. M. Bozga, R. Iosif, and F. Konečný. Deciding conditional termination. In *TACAS*, 2012.
6. M. Brain, T. Crick, M. D. Vos, and J. Fitch. TOAST: Applying answer set programming to superoptimisation. In *ICLP*, 2006.
7. C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *POPL*, 2009.
8. B. Cook, S. Gulwani, T. Lev-Ami, A. Rybalchenko, and M. Sagiv. Proving conditional termination. In *CAV*, 2008.
9. P. Cousot, R. Cousot, M. Fähndrich, and F. Logozzo. Automatic inference of necessary preconditions. In *VMCAI*, 2013.
10. L. de Moura and N. Bjørner. Z3: an efficient SMT solver. In *TACAS*, 2008.
11. E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, Aug. 1975.
12. B. Godlin and O. Strichman. Inference rules for proving the equivalence of recursive procedures. *Acta Inf.*, 45(6):403–439, July 2008.
13. B. Goldberg, L. Zuck, and C. Barrett. Into the loops: Practical issues in translation validation for optimizing compilers. *Electron. Notes Theor. Comp. Sci.*, 132, 2005.
14. S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *PLDI*, 2011.
15. S. Gulwani, S. Srivastava, and R. Venkatesan. Constraint-based invariant inference over predicate abstraction. In *VMCAI*, 2009.
16. S.-y. Guo and J. Palsberg. The essence of compiling with traces. In *POPL*, 2011.
17. C. Hawblitzel, M. Kawaguchi, S. K. Lahiri, and H. Rebêlo. Towards modularly comparing programs using automated theorem provers. In *CADE*, 2013.
18. R. Joshi, G. Nelson, and Y. Zhou. Denali: A practical algorithm for generating optimal code. *ACM Trans. Program. Lang. Syst.*, 28(6):967–989, Nov. 2006.
19. U. Junker. QUICKXPLAIN: Preferred explanations and relaxations for over-constrained problems. In *AAAI*, 2004.
20. S. Kundu, Z. Tatlock, and S. Lerner. Proving optimizations correct using parameterized program equivalence. In *PLDI*, 2009.
21. K. R. M. Leino. Efficient weakest preconditions. *Inf. Process. Lett.*, 93(6):281–288, Mar. 2005.
22. S. Lerner, T. Millstein, and C. Chambers. Automatically proving the correctness of compiler optimizations. In *PLDI*, 2003.
23. S. Lerner, T. Millstein, E. Rice, and C. Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *POPL*, 2005.
24. X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, July 2009.
25. N. P. Lopes and J. Monteiro. Automatic equivalence checking of UF+IA programs. In *SPIN*, 2013.
26. J. Marques-Silva, M. Janota, and A. Belov. Minimal sets over monotone predicates in boolean formulae. In *CAV*, 2013.
27. Y. Moy. Sufficient preconditions for modular assertion checking. In *VMCAI*, 2008.
28. S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
29. K. S. Namjoshi and L. D. Zuck. Witnessing program transformations. In *SAS*, 2013.
30. G. C. Necula. Translation validation for an optimizing compiler. In *PLDI*, 2000.
31. A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *TACAS*, 1998.
32. E. R. Scherpelz, S. Lerner, and C. Chambers. Automatic inference of optimizer flow functions from semantic meanings. In *PLDI*, 2007.

33. M. N. Seghir and D. Kroening. Counterexample-guided precondition inference. In *ESOP*, 2013.
34. M. Stepp, R. Tate, and S. Lerner. Equality-based translation validator for LLVM. In *CAV*, 2011.
35. R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation: a new approach to optimization. In *POPL*, 2009.
36. R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Generating compiler optimizations from proofs. In *POPL*, 2010.
37. Z. Tatlock and S. Lerner. Bringing extensibility to verified compilers. In *PLDI*, 2010.
38. J.-B. Tristan, P. Govereau, and G. Morrisett. Evaluating value-graph translation validation for LLVM. In *PLDI*, 2011.
39. X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *PLDI*, 2011.
40. A. Zaks and A. Pnueli. CoVaC: Compiler validation by program analysis of the cross-product. In *FM*, 2008.
41. J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic. Formal verification of SSA-based optimizations for LLVM. In *PLDI*, 2013.
42. L. Zuck, A. Pnueli, B. Goldberg, C. Barrett, Y. Fang, and Y. Hu. Translation and run-time validation of loop transformations. *Form. Methods Syst. Des.*, 27, 2005.