Incremental Verification of Compiler Optimizations*

Grigory Fedyukovich[1], Arie Gurfinkel[2], and Natasha Sharygina[1]

[1] University of Lugano, Switzerland,
{grigory.fedyukovich,natasha.sharygina}@usi.ch
[2] SEI/CMU, USA, arie@cmu.com

Abstract Optimizations are widely used along the lifecycle of software. However, proving the equivalence between original and optimized versions is difficult. In this paper, we propose a technique to incrementally verify different versions of a program with respect to a fixed property. We exploit a safety proof of a program given by a safe inductive invariant. For each optimization, such invariants are adapted to be a valid safety proof of the optimized program (if possible). The cost of the adaptation depends on the impact of the optimization and is often less than an entire re-verification of the optimized program. We have developed a preliminary implementation of our technique in the context of Software Model Checking. Our evaluation of the technique on different classes of industrial programs and standard LLVM optimizations confirms that the optimized programs can be re-verified efficiently.

1 Introduction

Program verification is necessary for building reliable software intensive systems. One challenge in using verification is deciding on the right level of abstraction. On one hand, verifying the source code (or a high-level compiler representation) before optimizations gives meaningful verification results to the user. On the other, verifying the binary (or a low-level compiler representation) after optimizations takes compiler out of the trusted computing base. Our experience with the UFO [1] indicates that verification results at both levels are desired.

There are two common techniques for adapting verification results from an original program P to an optimized program Q: (1) complete re-verification of Q; (2) establish property preserving equivalence (typically a form of a simulation) between P and Q. Re-verification is computationally expensive. Establishing a simulation between P and Q often requires manual instrumentation of the compiler which is hard to do and maintain [6]. In this paper, we propose an alternative solution that combines the advantages of the two approaches.

We assume that the original program P comes with a property G, and that P satisfies G (i.e., $P \models G$). Instead of showing an equivalence between P and Q,

* This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Department of Defense. This material has been approved for public release and unlimited distribution. DM-0000784

we show that Q satisfies G. First, by adapting the proof of $P \models G$, given by an inductive invariant, to Q, and then strengthening it by re-verification as needed. Our technique can be seen as a *property-specific equivalence*: P and Q are equivalent iff they both satisfy G.

We evaluate our approach on the `instcombine` optimization of LLVM that does local optimizations (such as turning `x = 1 + 1` into `x = 2`). Our experiments show that the approach is very effective. In many cases, the complete safety proof can be transferred between the original and the optimized programs. Whenever re-verification was required, it was insignificant.

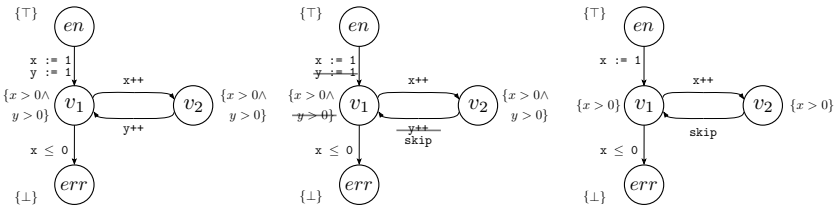
2 From Optimization to Evolution

In this section, we formally define the problem of *incremental property-directed verification of optimizations*. We begin with formal definitions of programs, safety proofs, and admissible optimizations. A "large-block" representation of a *program* is a tuple $P = (V, en, err, E, \tau_P)$, where V is a set of cutpoints (i.e., locations which represent heads of some loops); $en, err \in V$ are designated entry and error locations, respectively; $E \subseteq V \times V$ is the control-flow relation (represent a loop-free program fragments), $\tau_P : E \to \mathit{Stmt}^*$ maps control edges to loop-free program fragments. An example of a program is shown in Fig. 1a. We call the graph (V, E) the Cut-Point Graph (CPG) which collapses the more fine grained Control-Flow Graph (CFG).

We write $\vdash X$ to mean that X is valid. Let Expr be a set of expressions over program variables, $\mathit{pre}, \mathit{post} \in \mathit{Expr}$ and S a loop-free program fragment. We write $\vdash \{\mathit{pre}\}\ S\ \{\mathit{post}\}$ to mean that pre and post are pre- and post-conditions of S, i.e., whenever S starts in a state satisfying pre, if S terminates, it ends in a state satisfying post. A *safety proof* of P is a mapping $\psi : V \to \mathit{Expr}$ such that

$$\forall (u, v) \in E \cdot \vdash \{\psi(u)\}\ \tau_P(u, v)\ \{\psi(v)\} \qquad \psi(err) \to \bot \tag{1}$$

An *optimization* of P is a program $Q = (V, en, err, E, \tau_Q)$, that differs from P only in the labeling of the edges of the CPG. Note that this definition limits optimizations to changes that do not affect the loop-structure of P (i.e., loops cannot be added or removed). However, since we deal with "large-blocks", loop



(a) Program P and safety proof ψ.

(b) Program Q.

(c) Program Q and safety proof φ.

Figure 1: A simple program and an optimization.

Algorithm 1: optVerify(P, ψ, Q)

Input: *Original* program P, safety proof ψ of P, *new* program Q

Output: A pair $\langle res, \varphi \rangle$ s.t. $res \rightarrow \varphi$ is safety proof of Q

1 $\sigma \leftarrow$ guessMap(P, Q) ▷ Guess a map between variables of P and Q

2 $\pi \leftarrow$ mkInd$(\psi\sigma, Q, P)$ ▷ Weaken candidate $\psi\sigma$ until it is inductive for Q

3 $res, \varphi \leftarrow$ Verify(Q, π) ▷ Strengthen φ until it is SAFE for Q

4 **return** $\langle res, \varphi \rangle$

unrolling is admissible. At the same time, we do not put any other restrictions on Q. In particular, we do not require for Q to simulate P. An example of optimization Q of P is shown at the Fig. 1b in which the variable y was removed

The problem of incremental property-directed verification is: Given a program P, a safety proof ψ of P, and an optimization Q of P, *adapt* a safety proof ψ to a safety proof φ of Q, or show that Q does not admit a safety proof (i.e., has a counterexample).

Thus, we view the problem as a variant of *upgrade checking* (as opposed to verification). The key concept in upgrade checking is adapting verification results from one program to another. We extend this concept to safety proofs. Let P, ψ, and Q be as above. An adaptation π of φ to Q is a weakening of ψ such that

$$\forall w \in V \cdot \psi(w) \rightarrow \pi(w) \tag{2}$$

$$\forall (u, v) \in E \cdot \vdash \{\pi(u)\}\ \tau_Q(u, v)\ \{\pi(v)\} \tag{3}$$

Intuitively, (2) says that π is a weakening of ψ, and (3) says that π is inductive. Note that if $\pi(err) = \bot$, then π is a safety proof of Q. Otherwise, π is simply an inductive invariant.

3 Our Solution

optVerify is shown in Alg. 1. The input is an *original* program $P = (V, en, err, E, \tau_P)$, a safety proof $\psi : V \rightarrow Expr$ of P and a *new* program $Q = (V, en, err, E, \tau_Q)$. The output is a safety proof φ of Q or a counter-example. We require that P and Q share the CPG (i.e., same loop-heads), and differ only in the labeling of edges.

OptVerify consists of three steps: (1) a mapping σ between the variables of P and Q (line 1) is guessed and is used to syntactically transfer the safety proof ψ to a candidate proof $\psi\sigma$ of Q. (2) $\psi\sigma$ is weakened to π using mkInd until π is an inductive invariant (line 2). At this point, π is inductive, but not safe. (3) π is strengthened if necessary (and possible) to φ by a standalone verifier (line 3).

In our implementation, σ is guessed by syntactically matching variable names: a variable v of P is mapped to a variable v of Q if v exists in Q, and to a fresh symbol otherwise. To implement Verify, we use UFO [1] that iteratively strengthens the given inductive invariant. In particular, if π is already safe, UFO returns immediately and $\varphi \equiv \pi$. A verifier that cannot work by strengthening a given invariant would be useless as it would amount to verifying Q from scratch.

Algorithm 2: mkInd(ψ, Q, P)

Input: Candidate invariant ψ, *new* program Q, *original* program P

Output: Inductive invariant $\pi : V \to \mathit{Expr}$ of Q

```
1 π ← ψ; W ← {(u,v) ∈ E | τ_P(e) ≠ τ_Q(e)};
2 while W ≠ ∅ do
3     (u,v) ← getWtoSmallestEdge(W);            // according to the WTO
4     pre ← π(u) ; post ← π(v);
5     if (⊢ {pre} τ_Q(u,v) {post}) then W ← W \ {(u,v)};   // use SMT-solver
6     else
7         π(v) ← weakPost(pre, τ_Q(u,v), post);            // see Alg 3
8         W ← (W \ {(u,v)}) ∪ {(v,x) ∈ E | x ∈ V}
9 return π
```

Our key contribution is an algorithm mkInd for weakening a candidate invariant. We describe it in details in the rest of this section.

mkInd is shown in Alg. 2. The input is a candidate invariant ψ, a *new* program Q and an *original* program P. The output is an inductive weakening π of ψ (i.e., π satisfies (2) and (3)) for Q. mkInd maintains a work-list $W \subseteq E$ that is initialized with all the edges $(u, v) \in E$ on which P and Q disagree (i.e., $\tau_P(u, v) \neq \tau_Q(u, v)$). In each iteration of the main loop, first, an edge $(u, v) \in W$ that is least in the Weak Topological Ordering (WTO) [2] (in which inner loops are traversed before outer loops) is picked. Second, an SMT-solver is used to check whether the current values of $\pi(u)$ and $\pi(v)$ form a valid Hoare-triple for the corresponding loop-free code fragment $\tau_Q(u, v)$. If this is not the case, the post-condition $\pi(v)$ is weakened until the triple becomes valid and all outgoing edges of v are added to W. Soundness of mkInd is immediate – the work-list is empty only if every edge is annotated with a valid pre- and post-condition pair. Termination follows from the fact that at each iteration either the work-list is reduced, or a post-condition is weakened, and, our implementation of weakPost allows only for finitely many weakening steps.

weakPost is shown in Alg. 3. The input is a pre-condition *pre*, a post-condition *post* and a loop-free program fragment S. The output is a weakening *post*$'$ of *post* such that $\{\mathit{pre}\}\ S\ \{\mathit{post}'\}$ is valid. We assume that the post-condition *post* is given as a conjunction of lemmas, i.e., $\mathit{post} = \bigwedge_i \ell_i$. The algorithm computes the (possibly empty) subset of $\{\ell_i\}$ that forms a valid post-condition.

The naive implementation of weakPost iteratively checks whether each ℓ_i is a post-condition. Instead, we use an incremental SMT solver to do this enumeration efficiently. We assume that in addition to the SmtSolve API, an SMT solver has the method SmtAssert to add constraints to the current context. First, we compute an SMT-formula that encodes the verification condition (VC) of S. We use helper method mkVc that implements VC-generation from [5]. Second, we construct *vc* that determines the validity of pre- and post-condition by conjoining the pre-condition and negation of the post-condition (line 2) to the VC. Note that the post-condition is asserted under assumptions, encoded by Boolean variables x_i such that lemma ℓ_i is active iff x_i is true. We then itera-

Algorithm 3: WEAKPOST($pre, S, post$)

Input: $pre, post \in Expr$; $post = \bigwedge_{i=0}^{n} \ell_i$; $S \in Stmt^*$

Output: $post' \in Expr$, such that $\vdash \{pre\}\ S\ \{post'\}$ is valid

```
1 let {x_i | 0 ≤ i ≤ n} be fresh Boolean variables; U ← {0, ..., n};
2 vc ← pre ∧ MKVC(S) ∧ ¬(⋀_{i=0}^{n} (x_i → ℓ_i));
3 SMTASSERT(vc);
4 while SMTSOLVE( ) = SAT do
5     M ← SMTMODEL( );
6     foreach {0 ≤ i ≤ n | M ⊨ x_i} do SMTASSERT(¬x_i); U ← U \ {i};
7 post' ← ∧{ℓ_i | i ∈ U};
8 return post'
```

tively check the validity of vc. In each iteration, if vc is satisfiable, we assert $\neg x_i$ to disable the corresponding lemma(s). This terminates eventually since there are finitely many lemmas and at least one is disabled at every iteration. The conjunction of all active lemmas is returned as $post'$.

To summarize we illustrate MKIND on an example from Figs. 1a-1b. Given P, its safety proof ψ and Q, MKIND first checks the validity of the edge $en \to v_1$, i.e., $\{\top\}$ `x:=1` $\{x > 0 \wedge y > 0\}$. It is clearly invalid, thus WEAKPOST is used to weaken $\pi(v_1)$ to $x > 0$. Next, the validity of the triple for the edge $v_1 \to v_2$ is checked, i.e., $\{x > 0\}$ `x++` $\{x > 0 \wedge y > 0\}$. Note that the post-condition constructed in the previous step is now used as the pre-condition. Again, the triple is invalid and the post-condition is weakened to $x > 0$. Finally, the edges $v_2 \to v_1$ and $v_1 \to err$ are checked, and MKIND terminates. The final inductive invariant π of Q is shown in Fig. 1c. In this case, π is also a safety proof ($\pi(err) \to \bot$, and, therefore, $\varphi \equiv \pi$), so no further strengthening is required.

4 Evaluation

We have implemented OPTVERIFY in the UFO framework, and evaluated it on the Software Verification Competition (SVCOMP'13) benchmarks and `instcombine` optimization of LLVM. For each benchmark (300 - 5000 lines of source code), we measured the verification time OV, time of OPTVERIFY (MKIND + VERIFY), and time to re-verify from scratch (UV). Out of 397 safe benchmarks, we chose 108 that had non-trivial original verification time (OV $> 1s$). Due to lack of space, results are available at `http://www.inf.usi.ch/phd/fedyukovich/optVerify.pdf`.

In all but 6 cases, re-verification (VERIFY) was insignificant ($< 1s$). Furthermore, in 67 cases the candidate invariant was already safe and no strengthening was needed. In 52 cases, OPTVERIFY was at least an order of magnitude faster than re-verification (UV). This highlights the benefit of the approach. In 34 cases, `instcombine` dramatically reduced verification time (from minutes to $< 1s$). This shows that sometimes it is better to verify the optimized program first and then adapt the results to the original one. While this can be done using OPTVERIFY, we have not done so yet.

5 Related Work

There is a large body of related work on incremental verification, verifying compilers, and translation validation. Here, we only survey some of the most relevant techniques. Our algorithm is inspired by recent advancements in upgrade checking [3] and witnessing compiler transformations [6].

The goal of upgrade checking is to adapt a verification result from an original program P to an *upgraded* program Q, where Q is obtained from P by changing some of its functions. Current techniques [3] apply to bounded programs with functions, and work by adapting function summaries (i.e., relationship between function's inputs and outputs) from P to Q. In contrast, we work on unbounded programs without functions and adapt safety proofs (i.e., safe inductive invariants). Extending our approach to deal with functions is an interesting direction for future work.

The goal of witnessing compiler transformations is to formally establish equivalence (namely, stuttering simulation) between an original program P and its optimization Q. Current technique [6] works by instrumenting every optimization pass to output a witness relating its input and output. In contrast, we are interested in an automated solution. In that, our work is more similar to translation validation [7]. However, we only require equivalence with respect to a given property. At the technical level, our use of CPGs makes the approach less sensitive to many restructuring optimizations such as loop unrolling. We expect that in practice the two approaches can be combined, extending our algorithm, for example, to handle optimizations changing CPG structure.

At its core, our solution is similar to Houdini [4] inference algorithm that constructs inductive invariant out of a set of candidate formulas. However, there are many technical differences. One being our use of incremental SMT solver to speed up the search. More importantly, we need to adapt candidates from one program to another. Since based on discarding some candidate lemmas, our current adaptation strategy is rather rough and might be replaced by a more accurate one which weakens formulas on logical level and guided by results on analysis of concrete optimizations. This remains a challenge for future work.

References

1. Albarghouthi, A., Gurfinkel, A., Chechik, M.: UFO: A Framework for Abstraction- and Interpolation-Based Software Verification. In: Proc. of CAV'12 (2012)
2. Bourdoncle, F.A.: Efficient Chaotic Iteration Strategies with Widenings. In: Proc. of FMPA'93. pp. 128–141. LNCS (1993)
3. Fedyukovich, G., Sery, O., Sharygina, N.: eVolCheck: Incremental Upgrade Checker for C. In: Proc. of TACAS'13. LNCS, vol. 7795, pp. 292–307 (2013)
4. Flanagan, C., Leino, K.R.M.: Houdini: an Annotation Assistant for ESC/Java. In: FME. pp. 500–517 (2001)
5. Gurfinkel, A., Chaki, S., Sapra, S.: Efficient Predicate Abstraction of Program Summaries. In: NASA Formal Methods. pp. 131–145 (2011)
6. Namjoshi, K.S., Zuck, L.D.: Witnessing program transformations. In: SAS (2013)
7. Necula, G.C.: Translation validation for an optimizing compiler. In: PLDI (2000)