

Program Analysis for Compiler Validation

Anna Zaks
New York University
251 Mercer Street
New York, New York
ganna@cs.nyu.edu

Amir Pnueli
New York University
251 Mercer Street
New York, New York
amir@cs.nyu.edu

ABSTRACT

Translation Validation is an approach of ensuring compilation correctness in which each compiler run is followed by a validation pass that proves that the target code produced by the compiler is a correct translation (implementation) of the source code. It has been previously shown that the problem of translation validation can be reduced to checking if a single system - the cross-product of the source and target, satisfies a specific property. In this paper, we show how to adapt the existing program analysis techniques in the setting of translation validation. In addition, we present a novel invariant generation algorithm which strengthens our analysis when the input programs contain dynamically allocated data structures. Finally, we report on the prototype tool that applies the developed methodology to verification of the LLVM compiler. The tool handles many of the classical intraprocedural compiler optimizations such as constant folding, reassociation, common subexpression elimination, code motion, dead code elimination, and others.

1. INTRODUCTION

Optimizing compilers are quite large applications and are bound to have bugs, some of which may alter the behavior of programs being compiled. In safety critical and high-assurance software, where the effort of program correctness verification is extensive, it is highly advisable to ensure that the transformations performed by a compiler preserve the semantics of a program. That is precisely the goal of Translation Validation (TV) [11] - it ensures that compiler transformations preserve program semantics. In essence, instead of attempting verification of a given compiler, each compiler run is followed by a validation pass that automatically checks if the target code, produced by the compiler, is semantically equivalent to the source code. The Compiler Verification by Program Analysis of the Cross-Product (CoVaC) framework is a two-step solution to the program equivalence problem. First, one has to construct a *comparison system* that represents simultaneous execution of the

source and target programs. Second, one has to check if the comparison system satisfies a given specification. The general framework has been described in [16] along with the algorithm for the comparison system construction. Unlike the other translation validation frameworks [17, 10, 13], CoVaC does not rely on any compiler input (such as the compiler debugging information). In order to make the validator of non-cooperative compilers feasible and effective, the set of optimizations under consideration is limited to intraprocedural optimizations in which each branch (or a loop) in the target program corresponds to a branch (or a loop) in the source program. Many of the classical compiler optimizations such as constant folding, reassociation, induction variable optimizations, common subexpression elimination, code motion, register allocation, instruction scheduling, and others fall into this category.

However, as described in [16], the completeness of the CoVaC framework as well as its effectiveness depends on the methods for generation of the comparison system invariants. As one of the benefits from following the CoVaC approach, we can choose any existing invariant generation technique developed for a single system and plug it into the compiler verification framework. In this paper, we describe what existing methods we found to be effective and how they can be used in the CoVaC setting. We also present a novel technique for generating the invariants required for checking equivalence of dynamically allocated data structures, as there were no existing suitable method. Finally, we report on the experimental results which have been obtained by applying the CoVaC tool to verification of optimizing transformations performed by LLVM 1.9 [8, 2] - a very aggressive open-source compiler.

The rest of the paper is organized as follows. Section 2 gives an overview of the CoVaC framework. Section 3 focuses on the main contributions of this paper. It shows how the existing program analysis techniques can be applied in the CoVaC framework. It also presents the novel approach to generating invariants required for support of optimizations that involve dynamically allocated data structures. Finally, Section 4 presents the experimental results. We discuss the related work in Section 5.

2. THE COVAC FRAMEWORK

In this section, we briefly describe the CoVaC equivalence checking framework, which is formally presented in [16]. The main idea behind CoVaC is that the problem of establishing correct translation is equivalent to construction of a cross-product (comparison) graph $C = S \boxtimes T$ and checking if C

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...5.00.

satisfies a set of correctness conditions.

2.1 Transition Graphs

Our model is similar to that presented in [12] for verification of procedural programs. A program (application) \mathcal{A} consists of $m + 1$ procedures: $MAIN, P_1, \dots, P_m$, where $MAIN$ represents the main procedure, and P_1, \dots, P_m are procedures which may be called from $MAIN$ or from other procedures. We use $P_i(\vec{x}, \&\vec{z})$ to denote the signature of a procedure. Here, call-by-value parameter passing method is used for \vec{x} , and call-by-reference is used for \vec{z} . A procedure may return a result by means of \vec{z} variables. We use \vec{y} to denote the typed variables of a module. $\vec{y} = (\vec{x}; \vec{z}; \vec{w})$, i.e. the variables in \vec{y} are partitioned into \vec{x} , \vec{z} , and \vec{w} , where \vec{x} and \vec{z} are the *input* parameters and \vec{w} denotes the *local* variables of the module.

Each procedure is presented as a *transition graph*. Nodes of the graph are connected by directed edges labeled by instructions. There are four types of instructions: guarded assignments, procedure calls, and read/write operations. Consider a procedure $P_i(\vec{x}; \&\vec{z})$ with $\vec{y} = (\vec{x}, \vec{z}, \vec{w})$. Let \vec{u} include variables from \vec{y} ; and $E(\vec{y})$ be a list of expressions over \vec{y} .

- A *guarded assignment* is an instruction of the form $c \rightarrow [\vec{u} := E(\vec{y})]$, where guard c is a boolean expression. When the assignment part is empty, we abbreviate the label to a pure condition $c?$.
- *Read* and *write* instructions are denoted by $read(\vec{u})$ and $write(\vec{u})$. They are used to express the interaction of the procedure with the outside world; e.g. I/O instructions.
- *Procedure call* instruction $P_k(E(\vec{y}), \vec{u})$ denotes a call to the procedure $P_k(\vec{x}_f; \&\vec{z}_f)$, passing input parameters $E(\vec{y})$ by value and \vec{u} by reference.

Transition graphs can be used to model programs in procedural languages. In order to construct a formal model of a program, we first choose a set of program cut points Υ such that at least one location in each branch (or loop) belongs to Υ and the locations right before and after each read/write and call instruction belong to Υ . Each procedure (or function) whose implementation is given is represented by a transition graph. We choose the set Υ of a procedure P_i to be the set of nodes for the corresponding transition graph. For every pair of locations n, m in Υ , if there exists a path π from n to m , which does not pass through any other cut point, we add edge (n, m) to the graph and label it by the instruction that summarizes the effect of executing the path π .

2.2 Witness Comparison Graph

Assume we are given two procedures \mathcal{S} and \mathcal{T} . The *comparison transition graph* $\mathcal{C} = \mathcal{S} \boxtimes \mathcal{T}$ represents a simultaneous execution of \mathcal{S} and \mathcal{T} . The comparison graph variables consist of the source and target variables. Each node of the graph is a pair of source and target nodes. Each edge of the graph is labeled by a pair of instructions of the same type (both should be either read, write, procedure calls, or assignments). Note that this implies that the reads and writes of the two systems are always performed in sync. The edge labels should be either exactly the same as the corresponding

labels of the input systems or, alternatively, an assignment in one of the systems may be coupled with an ϵ -transition (a skip) in the other. The latter signifies the lack of progress in one of the systems. In addition to the structural requirements, no computation of \mathcal{C} may contain an infinite sequence of source (or target) ϵ -transitions; thus, every computation of the comparison graph has the corresponding computations in both source and target. And in the other direction, each source and target computation must be represented in \mathcal{C} .

An example of a comparison graph is presented in Fig. 1. We use capital variables to denote the variables of the source and their lower case counterparts for the target. First, the source procedure increments Y by 25. Second, both the source and target read a number from an I/O device. Third, the target catches up with the source by incrementing y by 25. Finally, both systems print out the products $Y * X$ and $y * x$.

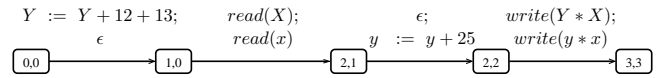


Figure 1: A comparison transition graph for $\mathcal{C}(\&(Y, y)) = \mathcal{S}(\&Y) \boxtimes \mathcal{T}(\&y)$.

A comparison graph \mathcal{C} is called a *witness of correct translation* if there exists a set of invariants $\{\varphi_l \mid l \in \text{nodes of } \mathcal{C}\}$ such that the following holds.

- For every edge e from node n to node m labeled by $(write(\vec{u}^S); write(\vec{u}^T))$,

$$\varphi_n \rightarrow (\vec{u}^S = \vec{u}^T).$$
- If n is the exit node of the comparison transition graph $\mathcal{S}(in : \vec{x}^S; \&\vec{z}^S) \boxtimes \mathcal{T}(in : \vec{x}^T; \&\vec{z}^T)$, we check if the values of the variables passed by reference are equal:

$$\varphi_n \rightarrow (\vec{z}^S = \vec{z}^T).$$

It has been shown in [16] that in order to check if \mathcal{T} is a correct translation of \mathcal{S} it is sufficient to:

1. construct a comparison graph $\mathcal{C} = \mathcal{S} \boxtimes \mathcal{T}$;
2. check if \mathcal{C} is a witness of correct translation.

2.3 Comparison Graph Construction

CoVaC framework can be used in various settings. In some cases, we may assume full knowledge of the inner workings of a particular compiler. For example, a self-certifying compiler may output a comparison graph. On the other hand, we may have to accommodate minimal (or no) compiler collaboration. Making the most liberal assumption is useful to users who may have to work with a particular existing compiler. It can also be of service to compiler developers to facilitate testing of immature compilers. [16] presents an algorithm for the comparison system construction that is suitable in the second setting - it only requires the source and the target procedures as its input. Here, we present a simplified version of the algorithm.

The algorithm is iterative and uses *WorkList* - a list of the comparison graph nodes, as the discovery frontier. The list is initialized with the procedure entry node (composed of the source and target procedure entry nodes). On each iteration,

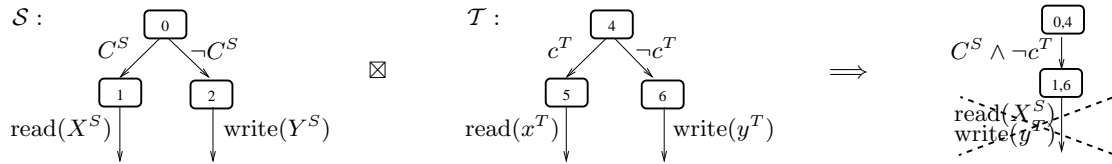


Figure 2: Motivation for branch alignment.

a node n from the *WorkList* is removed and new edges outgoing from n are discovered via composing the source and target edges. All the successors reachable by following the new edges are placed back into the *WorkList*. The following rules are used to match the source and target edges:

Rule 1: Only edges of the same type can be composed - both should be read, write, procedure call, or assignment edges. Guarded assignments are composed only if either both or none of the systems are currently at a branch node (or a loop head depending on the desired granularity). If only one of the systems can branch (execute a guarded assignment), it must wait for the other system to catch up using the second rule.

Rule 2: An ϵ -label can be matched up with an assignment; however, it is required that the ϵ -edge does not introduce an ϵ -cycle for any of the systems.

Rule 3: If none of the rules above are applicable, an error must be raised and the construction of \mathcal{C} should be aborted.

There is an obvious efficiency problem with *Rule 1* when we consider two nodes with multiple outgoing assignment edges. A straightforward approach, where we consider all possibilities (i.e., cartesian product), may lead to a number of edges in \mathcal{C} being quadratic to the number of edges in the input graphs. More importantly, if we mismatch the branches, unreachable nodes could be introduced into the graph, which may lead to further misalignment down the road. In particular, read, write, and function call edges may get out of sync. Consider the example in Fig. 2. Suppose $C^S = c^T$, $X^S = x^T$, and $Y^S = y^T$. Then one input program is a correct translation of the other. However, if we compose the edges $(0, 1)$ and $(4, 6)$ just relying on the fact that they are both conditional assignments, the algorithm presented so far will raise an error when examining the newly added unreachable node $\langle 1, 6 \rangle$. Thus, there is a need for comprehensive branch matching. One such method is presented below; in addition to resolving the misalignment issue, it usually constructs a comparison graph linear in the size of the input graphs.

Let \mathcal{C}_k be the partially constructed graph obtained after the k^{th} iteration of the algorithm. We can use the invariants $\{\varphi_l^k \mid l \in \text{nodes of } \mathcal{C}_k\}$ to facilitate the conditional branch alignment at iteration $k+1$. The invariants allow to rule out the matches that would introduce the infeasible paths into the comparison graph. Let φ_n^k be the invariant that holds at node (n^S, n^T) of graph \mathcal{C}_k . Let \mathcal{E}_n^S represent the set of source edges outgoing from n^S s.t. each edge $e_i^S \in \mathcal{E}_n^S$ is labeled by $c_i^S \rightarrow [u_i^S := E_i^S(\bar{y})]$. Similarly, we define \mathcal{E}_n^T - the set of target edges outgoing from n^T .

- A pair $(e_i^S, e_j^T) \in \mathcal{E}_n^S \times \mathcal{E}_n^T$ is matched if and only if
- it does not yet belong to the comparison graph and
 - $(\varphi_n^k \wedge c_i^S \wedge c_j^T)$ is satisfiable.

We only want to add an edge if there exists an execution through \mathcal{C}_k in which e_i^S and e_j^T are enabled simultaneously.

3. PRACTICAL INVARIANT GENERATION

The completeness and efficiency of the CoVaC approach heavily depends on invariant generation algorithms. The framework relies on auxiliary invariants to generate the comparison graph and to check if a generated graph is a witness of correct translation. We follow two strategies to obtain a practical solution. First, the assertions that are generated are goal oriented. In particular, it assumes that we only need to check for the validity of the formulas of the form $exp_1 = exp_2$. Second, we utilize a two-phase strategy where each phase provides a certain balance of precision and efficiency. In the first phase, we apply fast lightweight analysis. When it is not sufficient, we resort to deep and precise analysis. The overall work flow of the CoVaC tool is presented in Fig. 3.

3.1 Equivalence Checking

Instead of a general purpose invariant generation algorithm, CoVaC tool uses an oracle that checks if two input expressions are equivalent at a particular program location. Checking two expressions for equivalence is sufficient when confirming whether a graph is a witness of correct translation. We just need to ensure that at every node preceding the write instruction, the values that are being printed by the source and the target are the same. Another place where we need auxiliary invariants is branch alignment. We optimize the general approach and align branches by checking equivalence of the corresponding conditions instead of checking the satisfiability of the conjunctions as described at the end of Section 2. While this approach is less precise, it is still powerful enough to handle most classical compiler optimizations.

Each time we have to align the conditional assignments, we essentially match a branch instruction (or an if-statement) on the source with the one on the target. Assume the source edge e_+^S is taken when C^S holds; and e_-^S is taken when $\neg C^S$ holds. Similarly, there are two edges on the target: e_+^T and e_-^T , which are conditioned on c^T . Instead of checking the four formulas for satisfiability (following the method in Section 2), we use the fact that we are dealing with branch instructions, where the conditions are negations of each other, and consider the following cases:

- $(C^S \Leftrightarrow c^T)$ is valid - the conditions are equal; thus, the following edges are matched: (e_+^S, e_+^T) and (e_-^S, e_-^T) .
- $(C^S \Leftrightarrow \neg c^T)$ is valid - one condition is the negation of the other; the following edges are matched: (e_+^S, e_-^T) and (e_-^S, e_+^T) .
- Otherwise, we assume that the conditions are not related, so either all possible matches have to be made: (e_+^S, e_+^T) , (e_+^S, e_-^T) , (e_-^S, e_+^T) , and (e_-^S, e_-^T) , or we can use an ϵ -transition and freeze the execution of

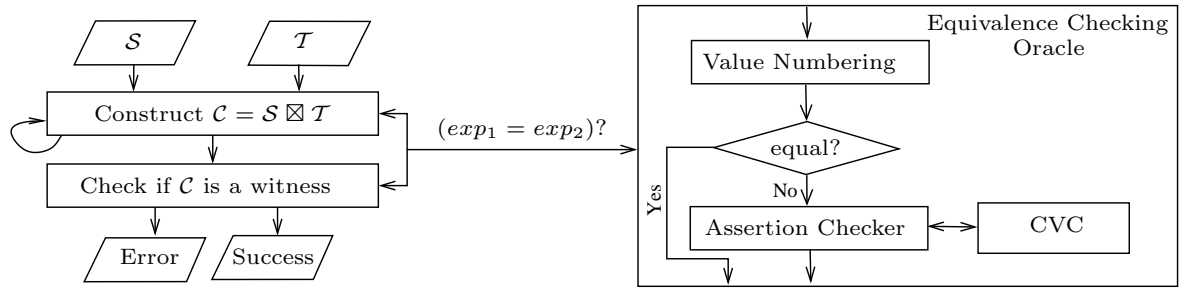


Figure 3: The Work Flow of the CoVaC tool.

the source system, obtaining the following matches: (e_+^S, ϵ) ; (e_-^S, ϵ) . The second option turns out to be better suitable in practice since it corresponds to an optimization in which both branches of an *if*-statement that assigns only to dead variables is removed by an optimizer.

The only case that we have not yet considered is when the conditions overlap. For example, $C^S = (x \geq 5)$ and $C^T = (x \geq 6)$. In this case, the set of the edges should be (e_+^S, e_+^T) ; (e_+^S, e_-^T) ; (e_-^S, e_-^T) . We would have to use the general matching rule to determine this dependency. However, the checks for such overlaps are rarely needed when dealing with proving translation in presence of compiler optimizations. The only exception is when one of the branches of a source *if*-statement is removed due to branch simplification. To address this optimization, we execute a pre-processing phase on both input programs in which we simplify the conditionals that evaluate to *true*.

3.2 Suitable Existing Techniques

In order to cover most common compiler optimizations, the algorithm has to reason in abstractions of uninterpreted functions and linear arithmetic. The problem of checking equality assertions in programs abstracted in the combined theory of linear arithmetic and uninterpreted functions, and whose conditionals are treated as non-deterministic, is coNP-hard [7]. Nevertheless, there exist efficient methods that are useful in determining the relationships between source and target expressions.

As depicted in Fig. 3, we employ a value numbering algorithm first. Global value numbering [9] assigns the same value number to provably equivalent variables and expressions throughout the procedure. This technique is particularly effective since we need an oracle to decide the validity of the formula $exp_1 = exp_2$. Value numbering is both fast and capable of detecting many value matches between the source and target expressions, especially, in the code fragments that have not been heavily optimized. Note that even when no optimizations are applied and the input systems are identical up to the renaming of the variables, there must be a technique in place capable of efficiently determining if the corresponding two variables are equal. We use the algorithm by Simpson [15], which provides a good balance between reasoning in theories of uninterpreted functions and linear arithmetic: it can detect a vast majority of equalities of expressions whose operators are treated as uninterpreted functions but also can easily handle simple constant folding and algebraic identities.

When value numbering is not strong enough to determine if two expressions are equivalent (due to excessive optimization), we resort to assertion checking - a static program verification technique based on computation of a weakest-precondition [5]. We generally follow methods like the one described in [3], for development of our assertion checker. A typical assertion checker (or a static program verifier) takes as an input a program and some assertion and generates from these a verification condition that implies the validity of the assertion in the program. The validity of the verification condition is checked by a theorem prover. We use CVC3 [1], an automatic theorem prover for the Satisfiability Modulo Theories, as a back end validity checker and use uninterpreted functions to represent the operators that are not supported by CVC3. As long as the compiler does not perform any simplifications based on the semantics of these operators, there is no loss of precision. The negative result - the expressions are not equal, is reported if we are unable to determine if two expressions are equivalent (for example, when the theorem prover is not strong enough to determine the validity of a verification condition). This ensures soundness of the method.

As a preprocessing step to the assertion checker, we simplify the input procedure based on the results of the value numbering: the same name is used to represent the variables with the same value number. This turns out to be crucial for both precision and efficiency of the assertion checker. Additional loop invariants, similar to those used in general purpose static verifiers [14], are used by the assertion checker. In addition to using the existing invariant generation techniques, we have developed the novel method that we use for proving equivalence of unbounded heaps. It is described in the next section.

3.3 Proving Equivalence of Unbounded Heaps

The program heap is modeled by unbounded arrays in CVC3 (Ex: ARRAY INT OF REAL), which allows to employ CVC3's theory of arrays. Consider the comparison system example in Fig. 4. Here, H_S and H_T denote the heaps of the source and the target programs. We assume that a and b are aliases. Since $H_S[b]$ is being assigned to by the edge (1, 3), the assignment to the source heap $H_S[a] := x$ is redundant and is removed in the target. The assignment $H_S[k] := i$ is also redundant since k is not updated within the loop and the value of $H_S[k]$ is altered by the edge (1, 3). In order to determine if the constructed graph is a witness, the assertion checker needs to determine if the values at the corresponding heap locations are equal: $(H_S[l] = H_T[l])$, for some address l . Since the heaps are dynamically updated

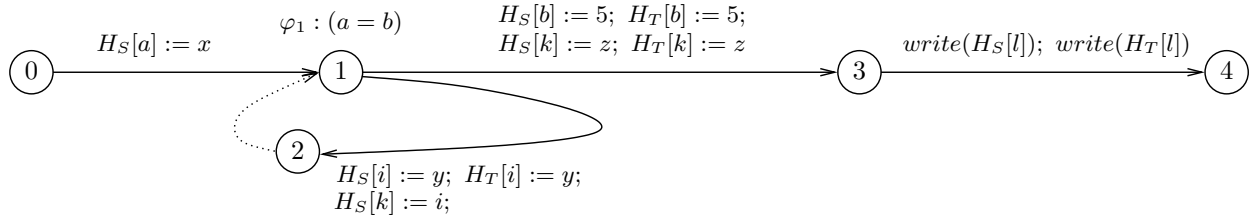


Figure 4: Heaps equivalence example. Only operations that involve H_S and H_T are shown. Source and target operations share the same variables as a result of value numbering.

within the loop, the number of locations which have to be considered can be unbounded. In addition, due to various optimizations like code motion and dead code elimination, the source and target heaps are not equal to each other at each node of the comparison program. Going back to our example, since the redundant stores $H_S[a] := x$ and $H_S[k] := i$ are eliminated in the target program, H_S may not be equal to H_T at the graph nodes 1 and 2. The key assumption we use is that, at each node of the comparison graph n , the heaps only differ from each other at a finite set of memory locations and the values at the rest of heap locations are equivalent. This assumption is fair in a setting of compiler validation.

For our analysis, we assume that the input comparison system is in SSA form [4]. Let \mathcal{N}_C denote the set of nodes of the comparison graph \mathcal{C} (which can be either a partially or a fully constructed graph). Next, we describe the procedure that computes $\Delta_n : n \in \mathcal{N}_C$ - the set of symbolic heap locations at which the heaps may possibly differ.

For every node n , Δ_n is initialized with \emptyset . Then, we iterate and at each iteration update the deltas according to the equation below. We stop when there is no change. In other words, the set of deltas is computed as the minimal fixed point of the equation.

Data Flow Equation: Let E_n be the set of edges incoming into node n . For an edge $e \in E_n$, let $head(e)$ denote the head node of e ; and let δ_e denote the set of heap locations that have been updated by the instructions of e .

$$\Delta_n := \Delta_n \cup \text{reduce} \left(\bigcup_{e \in E_n} \{ \Delta_{head(e)} \cup \delta_e \}, n \right)$$

For every edge e incoming into n , we add to the set Δ_n the locations at which the heaps may differ prior to executing the instructions of e and the locations that have been updated by e . Note that e may update H_S and H_T by storing the same expression at a location l . In that case, the $H_S[l] = H_T[l]$ at n and Δ_n should not include l . The *reduce* procedure removes the locations at which the heaps will become equal once we arrive at location n :

```

reduce ( SymbolicLocationsSet X_n, Node n )
for each l in X_n :
  if ( check_assertion(H_S[l] = H_T[l], n) )
    X_n = (X_n \ l);
return X_n;

```

In the pseudocode above, we use the assertion checker to determine if the values stored at the two heap locations are equal at node n . The assertion checker uses the invariants based on alias analysis and the invariants generated from the Δ_i , $i \in \mathcal{N}_C$ computed at the previous iteration. The invariant generation is described below.

Additional check has to be performed if the edge $e = (m, n)$ is a loop back edge. If any address from the set $\Delta_m \setminus \Delta_n$ is modified in the loop (a possibly different heap location is modified on each iteration of the loop), we report an error - the number of locations at which the heaps differ may be unbounded.

Invariant Generation: Given the computed Δ_n , we generate the following invariant for a node n :

$$\varphi_n = \forall i \in \mathbf{Z} \left(\bigwedge_{\forall l \in \Delta_n} i \neq l \rightarrow H_S[i] = H_T[i] \right)$$

Going back to the example from Fig. 4, below are the generated delta sets after each iteration.

	Δ_0	Δ_1	Δ_2	Δ_3	Δ_4
Initialization	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset
Iteration 1	\emptyset	$\{a\}$	$\{k\}$	\emptyset	\emptyset
Iteration 2	\emptyset	$\{a, k\}$	$\{a, k\}$	\emptyset	\emptyset
Iteration 3	\emptyset	$\{a, k\}$	$\{a, k\}$	\emptyset	\emptyset

Let's consider the second iteration of the algorithm. When considering node 1, $\Delta_1 = \{a\} \cup \text{reduce}(\delta_{(0,1)} \cup \Delta_2, 1) = \{a\} \cup \text{reduce}(\{a, k\}, 1) = \{a, k\}$. Next, node 2 is processed and we compute $\Delta_2 = \{k\} \cup \text{reduce}(\Delta_1 \cup \delta_{(1,2)}, 2) = \{k\} \cup \text{reduce}(\{a, i, k\}, 2) = \{a, k\}$. Since the edge $(2, 1)$ is a back edge, we check that k is not updated within the loop. When node 3 is processed, $\Delta_3 = \text{reduce}(\Delta_1 \cup \delta_{(1,3)}, 3) = \text{reduce}(\{a, k, b\}, 3) = \emptyset$. All the locations are removed by *reduce* since a and b are aliases, and the source and target heaps are overwritten with the same values at k and b . Finally, we compute $\Delta_4 = \text{reduce}(\Delta_3, 4) = \emptyset$. The computation stabilizes after three iterations.

The corresponding invariants can be encoded as the following predicates in CVC3:

$$\begin{aligned} \varphi_0 = \varphi_3 = \varphi_4 : & \text{FORALL } (i : INT) : (H_S[i] = H_T[i]) \\ \varphi_1 = \varphi_2 : & \text{FORALL } (i : INT) : ((i \neq a) \& (i \neq k)) \\ & \Rightarrow (H_S[i] = H_T[i]) \end{aligned}$$

Below is a more efficient version, which can also be used if the theorem prover does not support quantification:

$$\begin{aligned} \varphi_0 = \varphi_3 = \varphi_4 : & H_S = H_T \\ \varphi_1 = \varphi_2 : & ((H_S \text{ WITH } [a] := H_T[a]) \\ & \text{WITH } [k] := H_T[k]) = H_T \end{aligned}$$

Claim 1 If the algorithm terminates without an error, for every $n \in \mathcal{N}_C$, the generated φ_n is invariant at n .

PROOF. Assume that is not the case. Let the path π from the procedure entry r to some node n be a shortest counterexample. Then, there exists a heap location i , such that $H_S[i] \neq H_T[i]$ at n , while φ_n asserts otherwise. Meaning, there is no symbolic location $l \in \Delta_n$ that evaluates to i at n .

Consider the last time node n is processed. Suppose, the edge (v, w) is the last edge on the path π that assigned to the heap at location i . Then, there is a location $l \in \delta_{(v,w)}$ that evaluates to i . The location l will be propagated to n according to the data flow equation, unless it is reduced or the value of l is changed by a loop (the second would lead to an early termination with an error). Let's show that l cannot be reduced and thus belongs to Δ_n . Assume wrongly that $check_assertion(H_S[l] = H_T[l], u)$ returns *true* for some node u along the path from w to n . However, since $H_S[l] \neq H_T[l]$ at u for the execution π , it must be that one of the invariants associated with the nodes appearing on π from the beginning up to the last occurrence of u , but not including u , does not hold. Therefore, the counterexample π can be truncated starting from u , resulting in a shorter counterexample, which is a contradiction.

To finish the proof, we just need to show that l must evaluate to i at the last state of π . Note that l evaluates to i when we were taking the edge (v, w) . The value is unchanged since the procedure is in SSA form and l is not being assigned in a loop.

□

Claim 2 *The algorithm terminates.*

PROOF. Termination is guaranteed since the number of locations added to $\Delta_n : n \in \mathcal{N}_C$ monotonically grows; and the number of symbolic locations is limited by the number of program expressions. □

The number of iterations is bounded by $\mathcal{N}_C * c$, where c is the number of heap assignments. In practice, we rarely need to iterate for that long. First, we process the nodes in the topological order and use the most recently computed deltas, instead of the results obtained at the previous iteration. In addition, since loops usually have zero net effect on delta, it is uncommon that a node is processed more than twice.

Sound Treatment of Procedure Calls: Our analysis is intraprocedural. To ensure soundness, we check for the following:

- If an edge from a node n to a node m is a call to procedure foo , the procedure foo must not access the heaps at the locations in Δ_n . In fact, when dealing with compiler verification, either Δ_n is an empty set, or simple alias analysis are sufficient to check the condition.
- If a node r is the procedure exit node, $\Delta_r = \emptyset$. For the entry node t , it is assumed that $\Delta_t = \emptyset$ (Recall that the Δ_t is initialized with \emptyset and is never updated by the algorithm since the entry node does not admit any input edges). This condition ensures the zero net effect of the procedure. Consequently, for a call edge e , $\delta_e = \emptyset$.

4. EXPERIMENTAL RESULTS

We have constructed a prototype CoVaC tool based on the presented techniques and used it to verify the optimizations performed by LLVM compiler. LLVM [2] is an open source compiler for C and C++. We currently support a subset of C, which does not include function pointers, variable argument function calls, jumps.

We have tested the tool on a set of procedures compiled from the selected LLVM and CoVaC feature tests and third

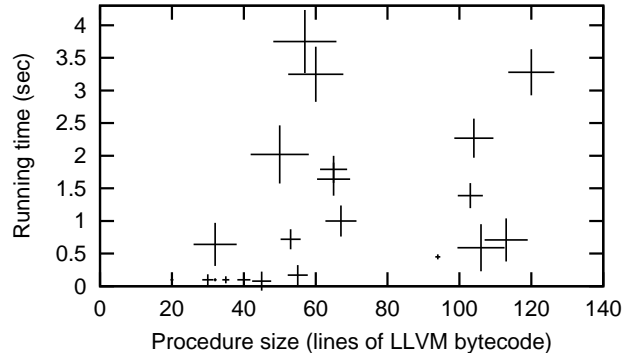


Figure 5: Dependency between the running time and the size of the procedure.

party implementation of classical algorithms like binary search, in-place heapsort, mergesort, Qsort, strcmp, primality testing, shortest paths, etc. Total line count is 2K of LLVM bytecode. On average, when validating highly optimized code (1/2 optimizations per line), CoVaC spends 0.02 seconds per every line of code. Fig. 5 shows the dependency of CoVaC tool running time on the procedure size. All of the selected benchmarks are the procedures that operate with dynamically allocated data structures. The size of the 'cross' is proportional to the number of optimizations performed. The most time is spent on assertion checking, which is dispatched once per every 8 lines when it is difficult to find a strong invariant with value numbering alone. This explains why the dependency of running time on the procedure size and the number of optimizations is not always consistent. We believe that the prototype's performance provides strong evidence that a practical validator can be constructed; taking into account that, unlike compiler, the tool is used few times per program's lifetime.

5. RELATED WORK

Good examples of the existing general translation validation frameworks that support a similar set of optimizations are [18], [13], and [10]. Though, [18] and [13] provide additional rules for loop reordering optimizations (loop interchange, fusion, etc.). All of the frameworks present program analysis and proof rules specialized to program equivalence checking and rely on the compiler debug annotations to guide their effort. For example, [18] uses the debug information to construct a set of candidate expressions that might be equal and then checks which of them are indeed equivalent. To facilitate the checking, [13] and [18] generate invariants over the variables of the *source* system based on the existing program analysis (like alias analysis) and specially developed techniques [6]. The approach of [10] is most similar to ours as it only relies on compiler annotations to predict the related if-statements. It presents a set of rewrite rules that are used to check if an expression of the source is equivalent to an expression on the target. In addition, [10] introduced the notion of memory equivalence in which the memory is equal except possibly at a finite set of heap locations.

6. REFERENCES

- [1] CVC3: An Automatic Theorem Prover for Satisfiability Modulo Theories (SMT).
- [2] The LLVM Compiler Infrastructure Project. <http://llvm.org>.
- [3] Mike Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In *PASTE*, 2005.
- [4] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct 1991.
- [5] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.
- [6] Yi Fang and Lenore D. Zuck. Improved invariant generation for TVOC. In *Proceedings of the 5th International Workshop on Compiler Optimization meets Compiler Verification*, 2006.
- [7] Sumit Gulwani and Ashish Tiwari. Assertion checking over combined abstraction of linear arithmetic and uninterpreted functions. In *The 15th European Symposium on Programming*, pages 279–293. Springer, March 2006.
- [8] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization*, 2004.
- [9] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
- [10] George C. Necula. Translation validation for an optimizing compiler. In *Programming Language Design and Implementation*, pages 83–95. ACM Press, 2000.
- [11] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. *Lecture Notes in Computer Science*, 1384:151–166, 1998.
- [12] Amir Pnueli. Verification of procedural programs. In *We Will Show Them! Essays in Honour of Dov Gabbay, Volume Two*, pages 543–590. College Publications, 2005.
- [13] Xavier Rival. Symbolic transfer function-based approaches to certified compilation. In *31st Symposium on Principles of Programming Languages*, pages 1–13. ACM Press, 2004.
- [14] S. Bensalem, Y. Lakhnech, and H. Saidi. Powerful techniques for the automatic generation of invariants. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102, pages 323–335, New Brunswick, NJ, USA, / 1996. Springer Verlag.
- [15] Loren Taylor Simpson. *Value-Driven Redundancy Elimination*. PhD thesis, Rice University, 1996.
- [16] Anna Zaks and Amir Pnueli. CoVaC: Compiler validation by program analysis of the cross-product. In *International Symposium on Formal Methods (FM 2008)*, Turku, Finland, May 2008.
- [17] Lenore Zuck, Amir Pnueli, Yi Fang, and Benjamin Goldberg. VOC: A methodology for the translation validation of optimizing compilers. *Journal of Universal Computer Science*, 9(3):223–247, 2003.
- [18] Lenore Zuck, Amir Pnueli, Benjamin Goldberg, Clark Barrett, Yi Fang, and Ying Hu. Translation and run-time validation of loop transformations. *Formal Methods in System Design*, 27(3):335–360, 2005.