Black-box equivalence checking across compiler optimizations

Manjeet Dahiya and Sorav Bansal

Indian Institute of Technology Delhi,
{dahiya, sbansal}@cse.iitd.ac.in

Abstract. Equivalence checking is an important building block for program synthesis and verification. For a synthesis tool to compete with modern compilers, its equivalence checker should be able to verify the transformations produced by these compilers. We find that the transformations produced by compilers are much varied and the presence of undefined behaviour allows them to produce even more aggressive optimizations. Previous work on equivalence checking has been done in the context of translation validation, where either a pass-by-pass based approach was employed or a set of handpicked optimizations were proven. These settings are not suitable for a synthesis tool where a *black-box* approach is required.
This paper presents the design and implementation of an equivalence checker which can perform black-box checking across almost all the composed transformations produced by modern compilers. We evaluate the checker by testing it across unoptimized and optimized binaries of SPEC benchmarks generated by `gcc`, `clang`, `icc` and `ccomp`. The tool has overall success rates of 76% and 72% for O2 and O3 optimizations respectively, for this first of its kind experiment.

1 Introduction

Equivalence checking is an important building block for program synthesis and verification. For a target sequence, the synthesis tool generates many possible optimized sequences and discharges the correctness check to the equivalence checker. The checker returns a proof of equivalence or it fails. Because the problem is undecidable in general, incorrect failures are inevitable, i.e., the equivalence checker can't be both sound and complete. In the setting of program synthesis and superoptimization, the checker must produce sound results: an incorrect equivalence failure (incompleteness) would result in a potentially missed optimization; on the other hand, a false positive (unsoundness) produces incorrect translation by the synthesis tool. Akin to compilers, a synthesis tool does not have test cases or traces given to it, requiring that the underlying equivalence checker be static. Another important difference from previous work on translation validation is that the equivalence checker is not aware of the exact nature of transformations performed. The transformation is a *black-box* to the checker, and the checker should be able to verify multiple composed transformations without knowing about the actual transformations or their sequence.

Previous work on equivalence checking has been performed in the context of translation validation. The goal of translation validation is to verify the correctness of a translation. This prior work has largely employed a pass-by-pass based approach [14,26], where each pass is verified separately by the equivalence checker, and/or worked with a set of handpicked transformations [14,26,37]. A pass-by-pass approach simplifies the verification process by dividing a big step into smaller and simpler steps, and the result is obtained by composing the results of individual steps. While this meets the objective of translation validation, these are unsuitable settings for a synthesis tool, where the nature and sequence of transformations are unknown. Note that an equivalence checker of a synthesis tool can be used for translation validation, while the converse is not true. In other words, the requirements on equivalence checking are stronger for synthesis than for translation validation. We also find that the underlying algorithms of previous techniques are not robust with respect to the transformations produced by modern compilers, e.g., Necula's TVI [26] fails when a simple `if`-block is replaced by a conditional move instruction (`cmov`). A detailed comparison with previous work is available in Sec. 6.

For a synthesis tool to compete with modern compilers, its equivalence checker should be able to verify the optimizations produced by these compilers. We find that the transformations produced by these compilers are much varied and presence of language level *undefined behaviour* allows them to produce even more aggressive optimizations. We present the design and implementation of an equivalence checker which meets the requirements of a synthesis tool and can verify the transformations produced by modern compilers. Our contributions towards this goal are:

– A new algorithm to determine the proof of equivalence across programs. The algorithm is robust with respect to modern compiler transformations and in a black-box manner, can handle almost all composed transformations performed by the modern compilers.
– New insights in equivalence checking, the most important being handling of language level undefined behaviour based optimizations. Previous work had disabled these optimizations, yet we find that these optimizations are very commonly used in compilers. For example, our equivalence checking success rates increase by 15%-52%, through modeling some important classes of undefined behaviour. To our knowledge, we are the first to handle undefined behaviour in equivalence checking for programs containing loops.
– Comprehensive experiments: we evaluate our implementation across black-box optimizations produced by modern compilers `gcc`, `clang`, `icc` (Intel's C Compiler), and `ccomp` (CompCert). Our tool can automatically generate proofs of equivalence, across O2/O3 compiler transformations, for 74% of the functions in C programs belonging to the SPEC benchmark suite across all four compilers. These results are comparable (and, in some cases, better) to the success rates achieved by previous translation validation tools which operated in much more restricted settings. This is a first of its kind experimental setup for evaluating an equivalence checker. We have also successfully

tested a preliminary superoptimizer supporting loops, with our equivalence checker.

2 Simulation relation as the basis of equivalence

Two programs are equivalent if for all equal inputs, the two programs have identical observables. We compute equivalence for C programs at function granularity. The inputs in case of C functions are the formal arguments and memory (minus stack) at function entry and the observables are the return values and memory (minus stack) at exit. Two functions are equivalent if for same arguments and memory (minus stack), the functions return identical return values and memory state (minus stack).

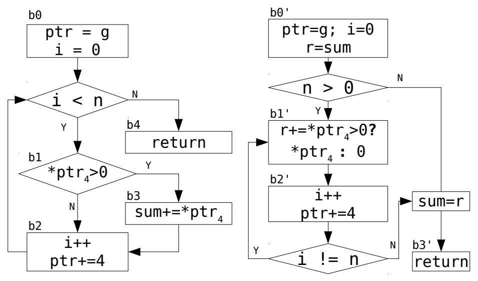
A simulation relation is a structure to establish equivalence between two programs. It has been used extensively in previous work on translation validation [14, 26, 28, 32, 41]. A simulation relation is a witness of the equivalence between two programs, and is represented as a table with two columns: Location and Relations. Location is a pair (L_1, L_2) of PC (program counter) in the two programs and relations are predicates (P) in terms of the variables at these respective PCs. The predicates P represent the invariants that hold across the two programs, when they are at the corresponding locations $L1$ and $L2$ respectively. A row $((L_1, L_2), P)$ of the simulation relation encodes that the relation P holds whenever the two programs are at L_1 and L_2 respectively. For a valid simulation relation, predicates at each location should be inductively provable from predicates at the predecessor locations. Further, the predicates at the entry location (pair of entry points of the two programs) must be provable using the input equivalence condition (base case). If the equivalence of the required observables is provable at the exit location (pair of exits of two programs) using the simulation relation predicates, we can conclude that the programs are equivalent.

Fig. 3a shows a simulation relation between the programs in Fig. 2a and Fig. 2b. It has three rows, one each for entry (b0, b0'), exit (b4, b3') and loop-node (b1, b1'). The predicates at entry and exit represent the equivalence of inputs (formal arguments and memory) and outputs (memory state (minus stack)) respectively. The predicates at the loop-node are required for the inductive proof of correctness of the simulation relation. The predicates represent the invariants which hold across the two programs, e.g., the predicate $i_A = i_B$ at loop-node represents that the "i" variables of the two programs, at b1 and b1' are equal. `Init` represents the input equivalence conditions and `Pre` represents the preconditions which can be assumed to be true for the programs; we model undefined behaviour through such preconditions. The only observable of this function is memory without the stack, as the function return type is `void`. The given simulation relation is valid and the predicates at the exit row can prove equivalence of observables, i.e., $M_A =_{\Delta} M_B$.

This simulation relation based technique can only prove equivalence across bi-similar transformations, e.g., it can not prove equivalence across the loop

```
int g[144]; int sum=0;
void sum_positive(int n) {
  int *ptr = g;
  for(int i = 0; i < n;
      i++, ptr++) {
    if (*ptr > 0)
      sum = sum + *ptr;
  }
}
```

Fig. 1: An example function accessing global variables g and sum. Undefined behaviour if n > 144 (our algorithm would capture this as preconditions).



(a) Unoptimized (b) Optimized

Fig. 2: Abstracted versions of unoptimized and optimized implementations of the program in Fig. 1. The ternary operator a?b:c represents the cmov assembly instruction.

tiling transformation. Fortunately, most compiler transformations preserve bisimilarity. Further, our notion of equivalence does not model constructs like non termination, exceptions, interrupts and concurrency.

In contrast with checking the correctness of a simulation relation, constructing the same is harder, and is in fact undecidable. The goal of our equivalence checking algorithm (Sec. 4.3) is to try and construct a valid simulation relation which can prove the equivalence. Before going into the details of the algorithm, we show its working for the example program of Fig. 1.

3 An illustrative example

Fig. 2a, 2b show the abstracted, unoptimized (A) and optimized (B) versions, of the program in Fig. 1. The optimized program has been compiled by gcc using -O3 flag. While the programs are in x86 assembly, we have abstracted them into C like syntax and flow charts for readability and exposition. The program has undergone multiple optimizations like 1) loop inversion, 2) condition inequality $(i < n)$ to condition disequality $(i \neq n)$ conversion, 3) usage of conditional move instruction (cmov) to get rid of a branch, and 4) register allocation (sum is written only once outside the loop). The last optimization is interesting, as it assumes that ptr cannot alias with the pointer sum. Notice that n is unknown and in general it is possible for ptr $\in$ [g, g+4*n) to overlap with sum. And hence, as such, the register allocation of global variable is not correct across the loop. However, compilers commonly apply such transformations by relying on aliasing based undefined behaviour assumptions. C specifies that an access beyond the size of a variable is undefined behaviour. In this example, the semantics restrict the value of ptr to lie within [g, g+4*144) and thus the compiler is free to assume that ptr cannot alias with sum.

In our knowledge, no previous work can handle this transformation. All of the previous work fail in proving this transformation correct, either due to one or multiple optimizations (from the above four). There are three broad improvements we make over previous work: 1) A robust algorithm for finding the correlation of program points. Our algorithm is the first to be demonstrated to work across black-box compiler transformations. 2) We present a systematic guess and check based inference of predicates without assumptions on the transformations performed. Our careful engineering of the guessing heuristics to balance efficiency and robustness is a novel contribution, and we evaluate it through experiments. Previous translation validation approaches, which can make more assumptions on the nature of transformations, did not need such a robust predicate inference procedure. 3) We model C level undefined behaviour conditions. Previous work on translation validation disabled transformations which exploit undefined behaviour.

Our goal is to first model the undefined behaviour (preconditions) and infer the simulation relation. Among all types of C undefined behaviour, aliasing based undefined behaviour is perhaps the most commonly exploited by compilers for optimization. In Fig. 1, the assumption that `ptr` and `sum` cannot alias with each other is an example of aliasing based undefined behaviour. To model such behaviour, we first need to reconstruct aliasing information from the compiled code. The details on our alias analysis algorithm and generation of the related undefined behaviour assumptions are available in [4]. In Fig. 1, alias analysis determines that `ptr` may alias with global variable `g`, and as per C semantics, `ptr` must always point within the region of `g`. This is represented by preconditions: (`ptr`$\geq$`g`) and (`ptr<g+144*4`). These preconditions are then used as assumptions while discharging proof obligations at the time of determining the correlation, and during the final simulation relation proof.

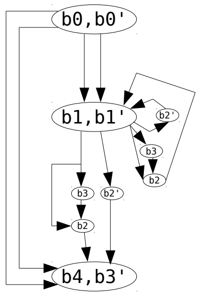
We now discuss how our algorithm computes a valid simulation relation; a simulation relation is represented using a *joint transfer function graph* (JTFG) (Sec. 4.2) which is constructed incrementally at each step. A JTFG represents a correlation across nodes and edges of the two programs. A JTFG node represents two PC values, one belonging to the first program and the other to the second program. Similarly, a JTFG edge represents one control flow edge in the first program and its correlated edge in the second program. Further, we assume that for two edges to be correlated in a JTFG, they should have equivalent *edge condition*, i.e., if one program makes a certain control transfer (follows an edge), the other program will make a corresponding control transfer along the respective correlated edge in the JTFG, and vice-versa. The individual edges of an edge of a JTFG could be composite: a composite edge (Sec. 4.2) between two nodes is formed by composing a sequence of edges (into a *path*), or by combining a disjunction of multiple paths (an example of a composite edge involving a disjunction of multiple paths is available in the following discussion).

Determining the correlation across program points and control transfers, is one of the trickiest problems during the construction of a simulation relation. Our algorithm proceeds as follows. We first fix the program points (PCs) and composite edges in one program (say $Prog_B$) and try to find the respective correlated program points and composite edges in the other program ($Prog_A$).

Location	Relations (P)
(b0,b0')	$n_A = n_B, g_A = g_B, sum_A = sum_B, M_A =_\Delta M_B$
(b1,b1')	$sl_4(M_A, sum_A) = r_B, n_A = n_B, i_A = i_B, sl_4(M_A, ptr_A) = sl_4(M_B, ptr_B), g_A = g_B, M_A =_{\Delta \cup sum_B} M_B, ptr_A = ptr_B, sum_A = sum_B, i_B + 1 \le n_B$
(b4,b3')	$M_A =_\Delta M_B$

Init: $n_A = n_B, g_A = g_B, sum_A = sum_B, M_A =_\Delta M_B$
Pre: $g_A \le ptr_A < g_A + 4 * 144$

(a) Simulation relation



(b) JTFG

Fig. 3: Simulation relation (JTFG) for the TFGs in Fig. 5. Table shows the predicates at each node and graph shows the correlation of edges and nodes of the two TFGs. `Init` is the initial conditions representing equivalence of inputs. `Pre` is the preconditions (undefined behaviour). $=_\Delta$ represents equivalent arrays except Δ; Δ represents the stack region. Operator sl_4 is a shorthand of *select* of size 4. The edges of the two programs between (b1,b1') to (b4,b3') and (b1,b1') are composite edges made up of the individual edges in between.

For sound reasoning of loops, we ensure that a correlation exists for at least one node in a loop, for all the loops. We pick the entry, exit and loop heads in $Prog_B$, as the *interesting* PCs that need to be correlated with PCs in $Prog_A$. In our example, we pick (b0', b1', b3') in $Prog_B$. Thus, $Prog_B$ can be represented as the three picked nodes, and a set of composite edges, $edges_B$ =(b0'-b1', b1'-b2'-b1', b1'-b2'-b3', b0'-b3'). We now try and find the correlated composite edges in $Prog_A$ for each composite edge in $Prog_B$. When all the composite edges of $Prog_B$ get correlated, we obtain a candidate correlation between the two programs.

Running our algorithm on the example, we initialize the JTFG with its entry node (b0, b0'). We pick an $edge_B$ from $edges_B$ (sorted in DFS order) and find the list of composite edges in $Prog_A$ (up to some fixed length) which can be correlated with $edge_B$. For edge (b0'-b1') of $Prog_B$ we get (b0-b1, b0-b4, b0-b1-b2, b0-b1-b3, b0-b1-b3-b2, b0-b1-b2||b0-b1-b3-b2)[1] as the list of potential composite edges in $Prog_A$, up to unroll factor 1 (unrolling the loop once). The last edge involves a disjunction of two paths. The conditions of these edges are $(0 < n_A, 0 \ge n_A, 0 < n_A \wedge *ptr_4 \le 0, 0 < n_A \wedge *ptr_4 > 0, 0 < n_A \wedge *ptr_4 > 0, 0 < n_A)$ respectively. And the condition of the current $edge_B$ is $0 < n_B$. However, the edge conditions of the two programs cannot be compared because there is no relation between n_A and n_B. Before comparing the conditions across these two programs, we need to find predicates which relate the variables of the two programs at (b0, b0'). In this example, we require the predicate $n_A = n_B$ at (b0, b0') (we later discuss how to obtain such predicates). Predicate $n_A = n_B$

[1] a-b-c is sequential composition of edges a-b and b-c. a-b-c||a-d-c is parallel composition of edges a-b-c and a-d-c.

proves that the conditions of (b0-b1) and (b0'-b1') are equal, implying that the correlation is correct. We then try to correlate the next composite edge of $Prog_B$ until all the composite edges are correlated and a simulation relation (JTFG) is found which can prove the required equivalence. At each step, it is possible for multiple composite edges in $Prog_A$ to have the required edge condition for correlation, while only one (or a few of the choices) may yield a provable simulation relation. To handle this, our algorithm backtracks to explore the remaining choices for correlated edges (discussed later).

At each step of the algorithm, a partial JTFG gets constructed. For future correlation we need to infer the predicates at the nodes of the currently constructed partial JTFG. We use a guess-and-check strategy to infer these predicates. This is similar to previous work on invariant inference (Houdini [8]), except that we are inferring these invariants/predicates on the JTFG, while previous work used this strategy for inferring invariants of an individual program. This guess-and-check procedure is formalized in Sec. 4.3.

The constructed JTFG may be incorrect on several counts. For example, it is possible that the predicates inferred at intermediate steps are stronger than what is eventually provable, and hence we infer an incorrect correlation. An incorrect correlation would mean that we will fail to successfully correlate in future steps of the algorithm, or will finish with a simulation relation that cannot prove observable equivalence. To handle either of these cases of incorrect correlation, our algorithm backtracks to try other potential composite edges for correlation, unwinding the decisions at each step. In theory, the algorithm is exponential in the number of edges to be correlated, but in practice, backtracking is rare, especially if the candidate edges for correlation are heuristically prioritized. Our simple heuristic to minimize backtracking is to explore the composite edges in the order of increasing depth, up to a given maximum limit controlled by unroll factor (μ). This heuristic is based on the assumption that a majority of the compiler transformations do not perform unrolling, and can thus be proven at smaller depths. If the algorithm succeeds in finding a JTFG that proves observable equivalence at the exit node, we have successfully computed a provable simulation relation, and hence completed the equivalence proof.

4 Formalization and algorithm

4.1 Abstracting programs as transfer function graph

We need an abstract program representation as a logical framework for reasoning about semantics and equivalence. This abstraction is called the transfer function graph (TFG). A TFG is a graph with nodes and edges. Nodes represent locations in the program, e.g., program counter (PC). Edges encode the effect of the instruction and the condition under which the edge is taken. The state of the program consists of bitvectors and a byte-addressable array, representing registers and memory respectively.

A simplified TFG grammar is presented in Fig. 4. The TFG $\mathbb{T}$ consists of preconditions, inputs, outputs and a graph $\mathbb{G}$ with nodes and edges. A node is

$$\begin{array}{ll}
\mathbb{T} & ::= ([\varepsilon], [\varepsilon], [\varepsilon], \mathbb{G}([node], [edge])) \\
node & ::= pc(int) \mid exit(int) \\
edge & ::= (node, node, edgecond, \tau) \\
edgecond & ::= state \rightarrow \varepsilon \\
\tau & ::= state \rightarrow state \\
state & ::= [(string, \varepsilon)] \\
\varepsilon & ::= var(string) \mid nryop(\varepsilon, \varepsilon) \mid select(\varepsilon, \varepsilon, int) \mid store(\varepsilon, \varepsilon, int, \varepsilon) \mid uif([\varepsilon])
\end{array}$$

Fig. 4: Transfer function graph ($\mathbb{T}$).

named either by its PC location ($pc(int)$), or by an exit location ($exit(int)$); a TFG could have multiple exits. An edge is a four-tuple with from-node and to-node (first two fields), its edge condition *edgecond* (third field) represented as a function from state to expression, and its transfer function τ (fourth field). An expression ε could be a boolean, bitvector, byte-addressable array, or an uninterpreted function. Expressions are similar to standard SMT expressions, with a few modifications for better analysis and optimization (e.g., unlike SMT, `select` and `store` operators have an additional third integer argument representing the number of bytes being read/written). An edge is taken when its *edgecond* holds. An edge's transfer function represents the effect of taking that edge on the program state, as a function of the state at the from-node. A state is represented as a set of (string, ε) tuples, where the string names the state element (e.g., register name). Apart from registers and memory, the state also includes an "IO" element indicating I/O activity, that in our setting, could occur only due to a function call (inside the callee)[2]. A procedure's TFG will have an entry node, and a single return (exit) node.

The C function calls in programs are modeled as uninterpreted functions (`uif`) in TFGs. The `uif` inputs include the callee function's ID, signature and the IO element value. `uif` modifies the state as per function signature. All "outputs" of the function call (memory, return values, and IO) are modified through `uif`. Fig. 5 shows the TFGs of unoptimized and optimized versions of our running example program.

4.2 Joint transfer function graph

A joint transfer function graph (JTFG) is a subgraph of the cartesian product of the two TFGs. Additionally, each JTFG node has predicates (second column of simulation relation) representing the invariants across the two programs. Intuitively, a JTFG represents a correlation between two programs: it correlates the move (edge) taken by one program with the move taken by the other program, and vice-versa. Formally, a JTFG (J_{AB}) between TFG_A and TFG_B is defined

[2] In the programs we consider, the only method to perform I/O is through function calls (that may internally invoke system calls).

Edge	condition	$\tau(ptr, i, n, M) =$
b0-b1	$n > 0$	$(g, 0, n, M)$
b0-b4	$n \leq 0$	$(g, 0, n, M)$
b1-b2	$sl_4(M, ptr) \leq 0$	(ptr, i, n, M)
b1-b3	$sl_4(M, ptr) > 0$	(ptr, i, n, M)
b3-b2	*true*	$let\ v = sl_4(M, ptr) + sl_4(M, sum)$ $(ptr, i, n, st_4(M, sum, v))$
b2-b1	$i + 1 < n$	$(ptr + 4, i + 1, n, M)$
b2-b4	$i + 1 \geq n$	$(ptr + 4, i + 1, n, M)$

Edge	condition	$\tau(ptr, i, r, n, M) =$
b0'-b1'	$n > 0$	$(g, 0, sl_4(M, sum), n, M)$
b1'-b2'-b1'	$i + 1 \neq n$	$let\ u = sl_4(M, ptr)$ $(ptr+4, i+1, r+(u > 0?u : 0), n, M)$
b1'-b2'-b3'	$i + 1 = n$	$let\ u = sl_4(M, ptr)$ $let\ v = r + (u > 0?u : 0)$ $(ptr + 4, i + 1, v, n, st_4(M, sum, v))$
b0'-b3'	$n \leq 0$	$(g, 0, sl_4(M, sum), n, M)$

Fig. 5: TFGs of the unoptimized (top) and optimized programs, represented as a table of edges. The 'condition' column represents the edge condition, and τ represents the transfer function. Operators sl_4 and st_4 are shorthands of *select* and *store* of size 4. *sum* and g represent the addresses of globals `sum` and `g[]` respectively.

as:

$$
\begin{gathered}
TFG_A = (N_A, E_A), TFG_B = (N_B, E_B), J_{AB} = (N_{AB}, E_{AB}) \\
N_{AB} = \{n_{AB} | n_{AB} \in (N_A \times N_B) \wedge (\bigvee_{e \in outedges_{n_{AB}}} edgecond_e)\} \\
E_{AB} = \{(e_{u_A \to v_A}, e_{u_B \to v_B}) \ | \{(u_A, u_B), (v_A, v_B)\} \in N_{AB} \wedge \\
edgecond_{e_{u_A \to v_A}} = edgecond_{e_{u_B \to v_B}}\}
\end{gathered}
$$

Here N_A and E_A represent the nodes and edges of TFG_A respectively and $e_{u_A \to v_A}$ is an edge in TFG_A from node u to node v. The condition on n_{AB} (in N_{AB}'s definition) stipulates that the disjunction of all the outgoing edges of a JTFG node should be `true`. The two individual edges ($e_{u_A \to v_A}$ and $e_{u_B \to v_B}$) in an edge of JTFG should have equivalent edge conditions. The individual edge (e.g., $e_{u_A \to v_A}$) within a JTFG edge could be a composite edge. Recall that a composite edge between two nodes may be formed by composing multiple paths between these two nodes into one. The transfer function of the composite edge is determined by composing the transfer functions of the constituent edges, predicated with their respective edge conditions. We use the `ite` (if-then-else) operator to implement predication. Fig. 3b shows a JTFG for the programs in Fig. 5.

```
Function Correlate(TFG_A, TFG_B)
    jtfg ← InitializeJTFG(EntryPC_A, EntryPC_B)
    edges_B ← DfsGetEdges(TFG_B)
    proofSuccess = CorrelateEdges(jtfg, edges_B, $\mu$)

Function CorrelateEdges(jtfg, edges_B, $\mu$)
    if edges_B is empty then
        return ExitAndIOConditionsProvable(jtfg)
    end
    edge_B ← RemoveFirst(edges_B)
    fromPC_B ← GetFromPC(edge_B)
    fromPC_A ← FindCorrelatedFirstPC(jtfg, fromPC_B)
    cedges_A ← GetCEdgesTillUnroll(TFG_A,fromPC_A,$\mu$)
    foreach cedge_A in cedges_A do
        AddEdge(jtfg, cedge_A, edge_B)
        PredicatesGuessAndCheck(jtfg)
        if IsEqualEdgeConditions(jtfg) $\wedge$ CorrelateEdges(jtfg, edges_B, $\mu$) then
            return true
        else
            RemoveEdge(jtfg, cedge_A, edge_B)
        end
    end
    return false

Function IsEqualEdgeConditions(jtfg)
    foreach e in edges(jtfg) do
        rel ← GetSimRelationPredicates($e_{fromPC}$)
        if $\neg$ (rel $\implies$ ($e_{FirstEdgeCond} \iff e_{SecondEdgeCond}$)) then
            return false
        end
    end
    return true
```

Algorithm 1: Determining correlation. μ is the unroll factor.

4.3 Algorithm for determining the simulation relation

Our correlation algorithm works across black-box compiler transformations, which is the primary difference between our work and previous work. Algorithm 1 presents the pseudo code of our algorithm. Function `Correlate()` is the top-level function which takes the TFGs of the two programs, and returns either a provable JTFG or a proof failure. The JTFG (`jtfg`) is initialized with its entry node, which is the pair of entry nodes of the two TFGs. Then, we get the the edges ($edges_B$) of TFG_B in depth-first-search order by calling `DfsGetEdges()`. And finally, the initialized `jtfg`, $edges_B$, and μ are passed as inputs to the `CorrelateEdges()` function, which attempts to correlate each edge in $Prog_B$ with a composite edge in $Prog_A$.

`CorrelateEdges()` consumes one edge from $edges_B$ at a time, and then recursively calls itself on the remaining $edges_B$. In every call, it first checks whether all $edges_B$ have been correlated (i.e., the `jtfg` is complete and correct). If it is so, it tries proving the observables, through `ExitAndIOConditionsProvable()`, and returns the status of this call. However, if there are still some edges left for correlation (i.e., `jtfg` is not complete), we pick an edge ($edge_B$) from $edges_B$ and try to find its respective candidate composite edge for correlation in TFG_A. Because we are correlating the edges in DFS order, the from-node of $edge_B$ (say $\texttt{fromPC}_B$) would have already been correlated with a node in TFG_A (say $\texttt{fromPC}_A$). We next compute the composite edges originating at $\texttt{fromPC}_A$ in TFG_A, to identify candidates for correlation with $edge_B$. The function `GetCEdgesTillUnroll()` returns the list of all composite edges ($cedges_A$) which start at $\texttt{fromPC}_A$ with a maximum unrolling of loops bounded by μ (unroll factor). $cedges_A$ represents the potential candidates for correlation with $edge_B$. The unroll factor μ allows our algorithm to capture transformations involving loop unrolling and software pipelining.

We check every $cedge_A$ in $cedges_A$ for potential correlation in the `foreach` loop in `CorrelateEdges()`. This is done by adding the edge ($cedge_A$, $edge_B$) to the JTFG and checking whether their edge conditions are equivalent; before computing this equivalence however, we need to infer the predicates on this partial JTFG through `PredicatesGuessAndCheck()` (discussed next). These inferred predicates are required to relate the variables at already correlated program points across the two programs. If the edge conditions are proven equivalent (`IsEqualEdgeConditions()`) we proceed to correlate (recursive call to `CorrelateEdges()`) the remaining edges in $edges_B$. If the conditions are not equal or the recursive call returns false (no future correlation found) the added edge is removed (`RemoveEdge()`) from `jtfg` and another $cedge_A$ is tried. If none of the composite edges can be correlated, the algorithm backtracks, i.e., the current call to `CorrelateEdges()` returns false.

Predicates guess-and-check is an important building block of our algorithm and it is one of the elements that lend robustness to our algorithm. Previous work has relied on inferring a relatively small set of syntactically generated predicates (e.g., [26, 32]) which are usually weaker, and do not suffice for black-box compiler transformations. Like Houdini [8], we guess several predicates generated through a grammar, and run a fixed point procedure to retain only the provable predicates. The guessing grammar needs to be general enough to capture the required predicates, but cannot be too large, for efficiency.

Guess: At every node of the JTFG, we guess predicates generated from a set $\mathbb{G} = \{\ \star_A \oplus \star_B, M_A =_{\star_A \cup \star_B} M_B\ \}$, where operator $\oplus \in \{<, >, =, \leq, \geq\}$. $\star_A$ and $\star_B$ represent the program values (represented as symbolic expressions) appearing in TFG_A and TFG_B respectively (including preconditions) and $M_A =_X M_B$ represents equal memory states except the region X. The guesses are formed through a cartesian product of values in TFG_A and TFG_B using the patterns in $\mathbb{G}$. This grammar for guessing predicates has been designed to work well with the transformations produced by modern compilers, while keeping the proof obligation discharge times tractable.

Check: Our checking procedure is a fixed point computation which eliminates the unprovable predicates at each step, until only provable predicates remain. At each step, we try and prove the predicates across a JTFG edge, i.e., prove the predicates at the head of the edge, using the predicates at the tail of the edge, and the edge's condition and transfer function. Further, the preconditions (Pre) determined through our model of language level undefined behaviour are used as assumptions during this proof attempt. The predicates at the entry node of JTFG are checked using the initial conditions across the two programs, represented by `Init`. `Init` consists of predicates representing input equivalence (C function arguments and input memory state). At each step, the following condition is checked for every edge:

$$\underset{(X \to Y) \in edges}{\forall} (Pre \wedge edgecond_{X \to Y}) \Rightarrow (preds_X \Rightarrow pred_Y)$$

Here $preds_X$ represents the conjunction of all the (current) guessed predicates at node `X` and $pred_Y$ represents a guessed predicate at node `Y`. $edgecond_{X \to Y}$ is the edge condition for the edge `(X→Y)`. If this check fails for some guessed predicate $pred_Y$ at some node `Y`, we remove that predicate, and repeat.

Undefined behaviour assumptions Most undefined behaviour modeling is straightforward. Aliasing based undefined behaviour requires a detailed alias analysis, however. For every memory access, our alias analysis algorithm determines the variables with which the memory address *may* alias. If an address a may alias with only one variable v, we emit preconditions encoding that a must belong to the region allocated to v. If an address may alias with multiple variables (or could point within the heap), then the corresponding preconditions involve a disjunction over all the respective regions. Additional precondition clauses are generated to encode that these regions allocated to different variables (and heap) may not overlap with each other. These aliasing based undefined behaviour assumptions are critical for achieving reasonable success rates for black-box equivalence checking across compiler transformations (Sec. 5).

We have tested the algorithm and its implementation across transformations like loop inversion, loop peeling, loop unrolling, loop splitting, loop invariant code hoisting, induction variable optimizations, inter-loop strength reduction, SIMD vectorization, and software pipelining. On the other hand, it does not support loop reordering transformations that are not simulation preserving, such as tiling and interchange. Also, if the transformations involve a reduction in the number of loops (e.g., replacing a loop-based computation with a closed form expression), the algorithm, in its current form, may fail to construct the proof.

5 Implementation and Evaluation

We compile multiple C programs by multiple compilers at different optimization levels, for x86, to generate unoptimized (-O0) and optimized (-O2 and -O3) binary executables. We then harvest functions from these executable files and

Bench	Fun	UN	Loop	SLOC	ALOC	Globals
mcf	26	2	21	1494	3676	43
bzip2	74	2	30	3236	9371	100
ctests	101	0	63	1408	4499	53
crafty	106	5	56	12939	72355	517
gzip	106	1	66	5615	14350	212
sjeng	142	3	68	10544	38829	312
twolf	191	17	140	17822	84295	348
vpr	272	69	155	11301	44981	153
parser	323	2	240	7763	30998	223
gap	854	0	466	35759	177511	330
vortex	922	5	116	49232	167947	815
perlbmk	1070	65	271	72189	175852	561

Table 1: Benchmarks characteristics. Fun, UN and Loop columns represent the total number of functions, the number of functions containing unsupported opcodes, and the number of functions with at least one loop, resp. SLOC is determined through the sloccount tool. ALOC is based on gcc-O0 compilation. Globals represent the number of global variables in the executable.

reconstruct C-level information, necessary for modeling undefined behaviour assumptions and for performing equivalence checks. Once the functions are harvested and C-level information is reconstructed, we perform the equivalence checks between the functions from unoptimized (O0) and optimized (O2/O3) executables.

The high level C program information necessary for performing the equivalence checking and modeling undefined behaviour are global variables and their scope/type attributes, local stack, function declarations and function calls, and program logic (function body). We reconstruct the language level semantics from ELF executables by using certain (standard) ELF headers. We rely on the debug headers (`-g`), symbol table and relocation table (`-fdata-sections --emit-relocs`) for getting the required high level information. Debug headers contain information about the functions and their signatures. Symbol table provides the global variable name, address, size and binding attributes. The relocation headers allows precise renaming of addresses appearing in code, to respective global variable identifiers with appropriate offsets, ensuring that the different placement of globals in different executables are abstracted away. None of these flags affect the quality of generated code. All these flags (or equivalent) are available in `gcc`, `clang`, `icc` and `ccomp`. Our reconstruction procedures are identical for both O0 and O2/O3 executables. The difference is that while the reconstructed information from O0 is used for obtaining the high level C program specification, the reconstructed information from O2/O3 is used only to help with proof construction.

Several optimizations were necessary to achieve reasonable results for equivalence across assembly programs. We have developed custom simplification proce-

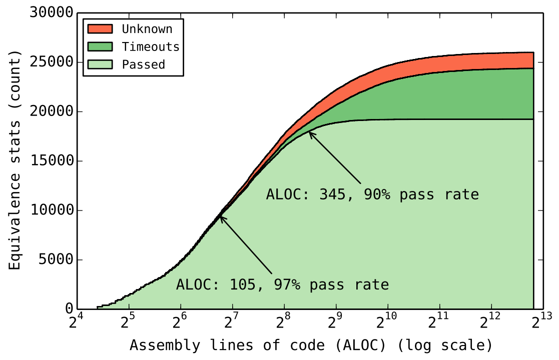


Fig. 6: Cumulative success rate (pass/fail) vs. ALOC.

Bench	% change
gzip-gcc2	26
gzip-clang2	15
bzip2-gcc2	44
bzip2-clang2	42
mcf-gcc2	52
mcf-clang2	46
parser-gcc2	43
parser-clang2	19

Fig. 7: The effect of modeling aliasing based undefined behaviour assumptions. The % change represents the difference between the success rates with and without modeling these undefined behaviour assumptions.

dures over expression trees to reduce expression size, for efficient pattern matching (e.g., for `select` and `store`) and efficient discharge of proof obligations. We use Z3 [5] and Yices [6] SMT solvers running in parallel for discharging proof obligations over our custom-simplified expressions, and use the result of solver which finishes first. We also employ caching of SMT query results to improve performance.

The programs that were compiled and checked, are listed in Table 1 along with their characteristics. `ctests` is a program taken from the CompCert testsuite and involves a variety of different C features and behaviour; the other programs are taken from the SPEC CPU2000 integer benchmarks. The SPEC benchmark programs do not include `gcc` and `eon` because their ELF executables files are very big, and our tool to harvest instruction sequences from executable files does not support such large ELF files. We also include an integer program from the SPEC CPU2006 integer benchmarks: `sjeng`. `sjeng` is one of the few C benchmarks in SPEC CPU2006 that is not already present in CPU2000, *and* has a low fraction of floating point operations. We avoid programs with significant floating-point operations, as our semantic models for x86 floating point instructions are incomplete. While compiling these programs, we disabled inlining, type based strict aliasing based assumptions, and signed integer strict overflow assumptions: our tool cannot handle interprocedural optimizations and does not model type based and signed integer overflow undefined behaviour assumptions.

A total of 4% of the functions contain unsupported (usually floating-point) opcodes (Tab. 1), and we plot results only for the functions which contain only supported opcodes. Fig. 6 plots the success rates as a function of the number of assembly instructions in a function, and Fig. 8 plots the success rates for each benchmark-compiler-optimization pair. There were 26007 function-pairs tested across all benchmarks and compiler/optimization pairs. The timeout value used was five hours. Overall, our tool is able to generate sound equivalence proofs

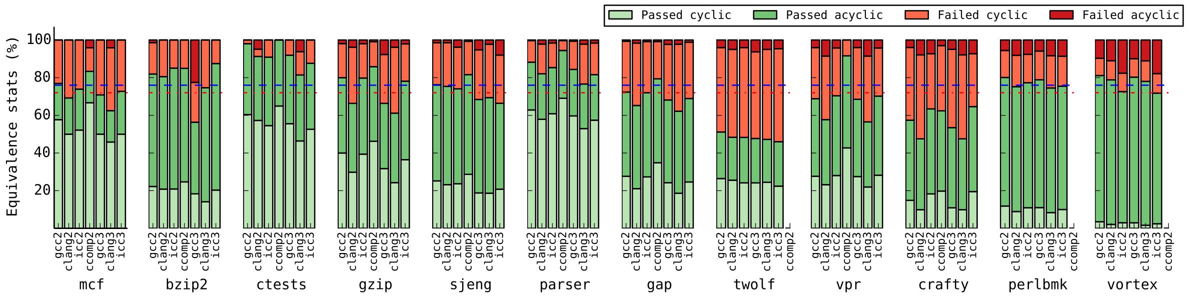


Fig. 8: Equivalence statistics. Functions with at least one loop are called "cyclic". The bar corresponding to a compiler (e.g., `clang`) represents the results across O0/O2 and O0/O3 transformations for that compiler (e.g., for `clang2` and `clang3` resp.). The average success rate across 26007 equivalence tests on these benchmarks, is 76% for O2 and 72% for O3 (dashed blue and red lines resp.). The missing bars for `ccomp` are due to compilation failures for those benchmarks.

across almost all transformations across multiple compilers for 76% of the tested function-pairs for O2 optimization level, and 72% of the tested function-pairs for O3 optimization level. The success rates are much higher for smaller functions, as seen in Fig. 6. The mean and median values for runtimes for passing equivalence tests are 313 seconds and 8.5 seconds respectively. 5% of the passing tests take over 1000 seconds to generate the result. Failures are dominated by timeouts (5 hours), inflating the mean and median runtimes for all (failing + passing) equivalence tests to 3962 seconds and 22 seconds respectively. The largest function for which equivalence was computed successfully has 4754 ALOC; the most complex function for which equivalence was computed successfully has 31 composite edges in its TFG_B.

Modeling undefined behaviour conditions is crucial for robustness. Tab. 7 shows the impact of modeling aliasing based undefined behaviour. If we do not model these undefined behaviour assumptions, the success rates decrease by 15%-52% for these programs.

Our experiments led to the discovery of one GCC bug (confirmed and fixed) [9] and two ICC bugs (one of them is confirmed, and the second involves confusion on the semantics of an ICC flag) [11, 12]. Each bug entails equivalence failures across multiple functions.

Finally, we have used our tool inside a 32-bit x86 brute-force superoptimizer that supports a rudimentary form of loops: it allows enumeration of straight-line instruction sequences potentially containing the x86 string instructions `scas`, `stos`, and `cmps` (the equivalent of `memchr`, `memset`, and `memcmp` functions, resp.); each of these instructions is modeled as a TFG containing a cycle. Through supporting these instructions, optimized implementations for common routines like initializing an array, and comparing elements of two arrays, get synthesized automatically, that are 1.04-12x faster than compiled code generated by any of the four compilers we discussed (across O2 and O3). In general, we expect the support for loops to enable general-purpose loop-based optimizations in a superoptimizer, and this work is an initial step towards this goal.

6 Related Work

One of the earliest examples of a translation validator can be found in [30]. Translation validation for mature compilers on large and complex programs, has been reported in at least two previous works: Translation validation infrastructure (TVI) [26] for GCC, and Value-graph translation validation [34, 38] for LLVM.

TVI demonstrated the validation of the gcc-2.91 compiler and the Linux-2.2 kernel, across five IR passes in GCC, namely branch optimization, common-subexpression elimination (CSE), loop unrolling and inversion, register allocation, and instruction scheduling. In TVI, validation is performed across each IR pass, i.e., first the input IR is validated against the output of the first pass, then the output of the first pass is validated against the output of the second pass, and so on. The TVI paper reports around 87% validation success rates. Necula's algorithm does not support loop unrolling, and that was reported as the primary cause for validation failures. There are several issues with TVI when applied to end-to-end (black-box and composed transformations) equivalence checking. First, this pass-based approach is not possible in a synthesis/superoptimizer setting. Second, TVI's heuristics for branch and memory-access correlations at basic-block granularity are syntactic, and fail for a large number of compiler transformations. Third, TVI relies on weakest-precondition based inference of simulation relation predicates, which is both expensive and less robust than our guessing procedure. For end-to-end checks, the substituted expressions generated by weakest-precondition become large and unwieldy, resulting in SMT solver timeouts. Further, guessing based on only weakest preconditions is often inadequate. Finally, TVI was tested across five compiler passes, and did not address several transformations, including those relying on undefined behaviour.

Value-graph translation validation for LLVM has been performed previously in two independent efforts [34, 38]. The value-graph based technique works by adding all known equality-preserving transformations for a program, to a *value graph*, until it saturates. Equivalence checking now involves checking if the graphs are isomorphic. In the work by Tristan et. al. [38], validation is performed across a known set of transformations, namely, dead-code elimination, global value numbering, sparse-condition constant propagation, loop-invariant code motion, loop deletion, loop unswitching, and dead-store elimination. Stepp et. al. [34] support all these transformations, and additionally enable partial-redundancy elimination, constant propagation, and basic block placement. While these tools capture several important transformations, they also omit many, e.g., loop inversion and unrolling, branch optimization, common-subexpression elimination, and instruction scheduling, to name a few. Some of these omitted transformations (e.g., loop inversion) enable more aggressive transformations, and so by omitting one of those, a chain of important transformation passes gets omitted. Also, none of these transformations rely on language-level undefined behaviour. For example, the transformations do not include the ones that could reorder accesses to global variables (e.g., by register-allocating them). Both papers report roughly 60-90% success rates for LLVM IR across the transformations they support. Compared head-to-head, this is comparable to our success rates, albeit in a much more restricted setting. A value-graph approach is limited by the vocabulary of transformations that are supported by the translation validator, and thus seems less general than constraint-based approaches like TVI and ours. Also, the number of possible translations for passes like register allocation and instruction scheduling is likely to grow exponentially in a value-graph approach. At least with the current evidence, it seems unlikely that the value-graph based translation validation approach would yield good results for end-to-end black-box equivalence checking.

Data-driven equivalence checking (DDEC) [32] is an effort perhaps closest to our goals of checking equivalence on x86 assembly programs. However, DDEC takes a radically different approach of relying on the availability of execution traces for high-coverage tests, an assumption that is not always practical in a general compiler optimization setting. DDEC was tested on a smaller set of examples (around 18) of x86 assembly code generated using GCC and CompCert, and all DDEC test examples are a part of our `ctests` benchmark. Compared head-to-head with DDEC, our algorithm is static (does not rely on execution traces), supports a richer set of constructs (stack/memory/global accesses, function calls, undefined behaviour), is more robust (tested on a much larger set of programs, and across a richer set of transformations), and more efficient (when compared head-to-head on the same programs). While DDEC can infer linear equalities through execution traces, it cannot handle several other types of non-linear invariants (e.g., inequalities) often required to prove equivalence across modern compiler transformations. Recent work on loop superoptimization for Google Native Client [3] extends DDEC by supporting inequality-based invariants; the evaluation however is limited to a small selection of test cases, and hence does not address several scalability and modeling issues that we tackle in our equivalence checker. For example, the authors do not model undefined behaviour, which we find is critical for black-box equivalence checking across real programs.

The *Correlate* module of parameterized program equivalence checking (PEC) [14] computes simulation based equivalence for optimization patterns represented as parameterized programs containing *meta-variables*. In contrast, we are interested in equivalence checking across black-box transformations involving low level syntax, as is typical in synthesis and superoptimization settings: our correlation algorithm with guessing procedures have been evaluated for this use case. In PEC's setting, the presence of meta-variables usually provides an easier correspondence between the two programs, greatly simplifying the correlation procedure; the relations (predicates relating variables in two programs) across meta-variables are also easier to determine in this setting.

Previous work on regression verification [7,35] determines equivalence across structurally similar programs, i.e., programs that are closely related, with similar control structure and only a small (programmer introduced) delta between the two programs. In our setting, the programs being compared are significantly different because of transformations due to multiple composed compiler optimizations. While our equivalence checker can correctly compute equivalence across all the examples presented in regression verification [7, 35], the converse is not true.

There are more approaches to translation validation and equivalence checking (e.g., [13,22,24,29,41,44]), and most have been evaluated on a variety of relatively smaller examples. To our knowledge, previous work has not dealt with compiler transformations in as much generality, as our work. Our work also overlaps with previous work on verified compilation [21, 42, 43], compiler testing tools [17, 18, 40], undefined behaviour detection [39], and domain specific languages for coding and verifying compiler optimizations [19, 20].

In terms of the correlation algorithm, our approach is perhaps closest to CoVaC [41], in that we both construct the JTFG incrementally, and rely on an invariant generation procedure, while determining the correlations. There are important differences however. CoVaC relies on an oracular procedure called *InvGen*; we show a concrete implementation of `PredicatesGuessAndCheck()`. Further, we differ significantly in our method to identify the correlations. CoVaC relies on correlating *types* of operations (e.g., memory reads and writes are different types), which is similar to TVI's syntactic memory correlations, and is less general than our semantic treatment of memory. Also, CoVaC relies on the *satisfiability* of the conjunction of edge conditions (viz. *branch alignment*) in the two TFGs, which is unlikely to work across several common transformations that alter the branch structure. CoVaC was tested on smaller examples across a handful of transformations. In contrast, our correlation method based on *equality* of condition of composite edges is more general, and we demonstrate this through experiments. Further, backtracking and careful engineering of guessing heuristics are important novel features of our procedure.

Most previous translation validation work (except DDEC) has been applied to IR. There has also been significant prior work on assembly level verification, through equivalence checking. SymDiff [10, 15, 16] is an effort towards verifying compilers and regression verification, and works on assembly code. However, the support for loops in SymDiff is quite limited — they handle loops by unrolling them twice. Thus, while SymDiff is good for checking partial equivalence, and to catch errors across program versions and translations, generation of sound equivalence proofs for programs with loops is not supported.

Alive [23] verifies *acyclic* peephole optimization patterns of the `InstCombine` pass of LLVM and models undefined behaviour involving undefined values, poison values and arithmetic overflow. While Alive computes equivalence across acyclic programs, we are interested in simulation based equivalence for programs with loops.

Program synthesis and superoptimization techniques [1, 2, 25, 27, 31, 33, 36] rely on an equivalence checker (verifier) for correctness. The capabilities of a synthesis based optimizer are directly dependent on the capabilities of its underlying equivalence checker. We hope that our work on black-box equivalence checking informs future work in program synthesis and superoptimization.

References

1. Bansal, S., Aiken, A.: Automatic generation of peephole superoptimizers. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 394–403. ASPLOS XII, ACM (2006)
2. Bansal, S., Aiken, A.: Binary translation using peephole superoptimizers. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation. pp. 177–192. OSDI'08, USENIX Association (2008)
3. Churchill, B., Sharma, R., Bastien, J., Aiken, A.: Sound loop superoptimization for google native client. In: Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 313–326. ASPLOS '17, ACM (2017)
4. Dahiya, M., Bansal, S.: Modeling undefined behaviour semantics for checking equivalence across compiler optimizations. In: Hardware and Software: Verification and Testing - 13th International Haifa Verification Conference, HVC 2017
5. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. TACAS'08/ETAPS'08
6. Dutertre, B.: Yices 2.2. In: Biere, A., Bloem, R. (eds.) Computer-Aided Verification (CAV'2014). Lecture Notes in Computer Science, vol. 8559, pp. 737–744
7. Felsing, D., Grebing, S., Klebanov, V., Rümmer, P., Ulbrich, M.: Automating regression verification. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering. pp. 349–360. ASE '14, ACM (2014)
8. Flanagan, C., Leino, K.: Houdini, an annotation assistant for esc/java. In: FME 2001: Formal Methods for Increasing Software Productivity, Lecture Notes in Computer Science, vol. 2021, pp. 500–517. Springer Berlin Heidelberg (2001)
9. GCC Bugzilla - Bug 68480, https://gcc.gnu.org/bugzilla/show_bug.cgi?id=68480
10. Hawblitzel, C., Lahiri, S.K., Pawar, K., Hashmi, H., Gokbulut, S., Fernando, L., Detlefs, D., Wadsworth, S.: Will you still compile me tomorrow? static cross-version compiler validation. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. pp. 191–201. ESEC/FSE 2013, ACM (2013)
11. ICC developer forums discussion: icc-16.0.3 not respecting fno-strict-overflow flag?, https://software.intel.com/en-us/forums/intel-c-compiler/topic/702516
12. ICC developer forums discussion: icc-16.0.3 not respecting no-ansi-alias flag?, https://software.intel.com/en-us/forums/intel-c-compiler/topic/702187
13. Kanade, A., Sanyal, A., Khedker, U.P.: Validation of gcc optimizers through trace generation. Softw. Pract. Exper. 39(6), 611–639 (Apr 2009)
14. Kundu, S., Tatlock, Z., Lerner, S.: Proving optimizations correct using parameterized program equivalence. In: Proceedings of the 2009 ACM SIGPLAN Conference

on Programming Language Design and Implementation. pp. 327–337. PLDI '09, ACM (2009)
15. Lahiri, S., Hawblitzel, C., Kawaguchi, M., Rebelo, H.: Symdiff: A language-agnostic semantic diff tool for imperative programs. In: CAV '12. Springer (2012)
16. Lahiri, S., Sinha, R., Hawblitzel, C.: Automatic rootcausing for program equivalence failures in binaries. In: Computer Aided Verification (CAV'15). Springer (2015)
17. Le, V., Afshari, M., Su, Z.: Compiler validation via equivalence modulo inputs. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 216–226. PLDI '14, ACM (2014)
18. Le, V., Sun, C., Su, Z.: Finding deep compiler bugs via guided stochastic program mutation. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. pp. 386–399. OOPSLA 2015, ACM (2015)
19. Lerner, S., Millstein, T., Chambers, C.: Automatically proving the correctness of compiler optimizations. In: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation. pp. 220–231. PLDI '03, ACM (2003)
20. Lerner, S., Millstein, T., Rice, E., Chambers, C.: Automated soundness proofs for dataflow analyses and transformations via local rules. In: Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 364–377. POPL '05, ACM (2005)
21. Leroy, X.: Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In: 33rd ACM symposium on Principles of Programming Languages. pp. 42–54. ACM Press (2006)
22. Leung, A., Bounov, D., Lerner, S.: C-to-verilog translation validation. In: Formal Methods and Models for Codesign (MEMOCODE). pp. 42–47 (2015)
23. Lopes, N.P., Menendez, D., Nagarakatte, S., Regehr, J.: Provably correct peephole optimizations with alive. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 22–32. PLDI 2015, ACM (2015)
24. Lopes, N.P., Monteiro, J.: Automatic equivalence checking of programs with uninterpreted functions and integer arithmetic. Int. J. Softw. Tools Technol. Transf. 18(4), 359–374 (Aug 2016)
25. Massalin, H.: Superoptimizer: A look at the smallest program. In: Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems. pp. 122–126. ASPLOS II, IEEE Computer Society Press (1987)
26. Necula, G.C.: Translation validation for an optimizing compiler. In: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation. pp. 83–94. PLDI '00, ACM (2000)
27. Phothilimthana, P.M., Thakur, A., Bodik, R., Dhurjati, D.: Scaling up superoptimization. In: Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 297–310. ASPLOS '16, ACM (2016)
28. Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In: Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems. pp. 151–166. TACAS '98, Springer-Verlag (1998)
29. Poetzsch-Heffter, A., Gawkowski, M.: Towards proof generating compilers. Electron. Notes Theor. Comput. Sci. 132(1), 37–51 (May 2005)

30. Samet, H.: Proving the correctness of heuristically optimized code. Commun. ACM 21(7), 570–582 (Jul 1978)
31. Schkufza, E., Sharma, R., Aiken, A.: Stochastic superoptimization. In: Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 305–316. ASPLOS '13, ACM (2013)
32. Sharma, R., Schkufza, E., Churchill, B., Aiken, A.: Data-driven equivalence checking. In: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications. pp. 391–406. OOPSLA '13, ACM (2013)
33. Sharma, R., Schkufza, E., Churchill, B., Aiken, A.: Conditionally correct superoptimization. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. pp. 147–162. OOPSLA 2015, ACM (2015)
34. Stepp, M., Tate, R., Lerner, S.: Equality-based translation validator for llvm. In: Proceedings of the 23rd International Conference on Computer Aided Verification. pp. 737–742. CAV'11, Springer-Verlag (2011)
35. Strichman, O., Godlin, B.: Regression verification - a practical way to verify programs. In: Verified Software: Theories, Tools, Experiments, vol. 4171, pp. 496–501. Springer Berlin Heidelberg (2008)
36. Tate, R., Stepp, M., Lerner, S.: Generating compiler optimizations from proofs. In: Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 389–402. POPL '10, ACM (2010)
37. Tate, R., Stepp, M., Tatlock, Z., Lerner, S.: Equality saturation: a new approach to optimization. In: POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages. pp. 264–276. ACM (2009)
38. Tristan, J.B., Govereau, P., Morrisett, G.: Evaluating value-graph translation validation for llvm. In: Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 295–305. PLDI '11, ACM (2011)
39. Wang, X., Zeldovich, N., Kaashoek, M.F., Solar-Lezama, A.: Towards optimization-safe systems: Analyzing the impact of undefined behavior. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. SOSP '13
40. Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and understanding bugs in c compilers. In: Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 283–294. PLDI '11, ACM (2011)
41. Zaks, A., Pnueli, A.: Covac: Compiler validation by program analysis of the cross-product. In: Proceedings of the 15th International Symposium on Formal Methods. pp. 35–51. FM '08, Springer-Verlag (2008)
42. Zhao, J., Nagarakatte, S., Martin, M.M., Zdancewic, S.: Formalizing the llvm intermediate representation for verified program transformations. In: Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 427–440. POPL '12, ACM (2012)
43. Zhao, J., Nagarakatte, S., Martin, M.M., Zdancewic, S.: Formal verification of ssa-based optimizations for llvm. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 175–186. PLDI '13, ACM (2013)
44. Zuck, L., Pnueli, A., Fang, Y., Goldberg, B.: Voc: A methodology for the translation validation of optimizing compilers 9(3), 223–247 (2003)