

Modeling undefined behaviour semantics for checking equivalence across compiler optimizations

Manjeet Dahiya and Sorav Bansal

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

Abstract. Previous work on equivalence checking for synthesis and translation validation has usually verified programs across selected optimizations, disabling the ones that exploit undefined behaviour. On the other hand, modern compilers extensively exploit language level undefined behaviour for optimization. Previous work on equivalence checking for translation validation and synthesis yields poor results, when such optimizations relying on undefined behaviour are enabled.

We extend previous work on simulation-based equivalence checking, by adding a framework for reasoning about language level undefined behaviour. We implement our ideas in a tool to compare equivalence across compiler optimizations produced by GCC and LLVM. Testing these compiler optimizations on programs taken from the SPEC integer benchmark suite, we find that modeling undefined behaviour semantics improves success rates for equivalence checking by 31 percentage points (from 50% to 81%) on average, almost uniformly across the two compilers. This significant difference in success rates confirms the widespread impact of undefined behaviour on compiler optimization, something that has been ignored by previous work on equivalence checking. Further, our work brings insight into the relative significance of the different types of C undefined behaviour on compiler optimization.

1 Introduction

Programming languages have erroneous conditions in the form of erroneous program constructs and erroneous data. Language standards do not impose requirements on all such erroneous conditions. The erroneous conditions on which no requirements have been imposed by the standard, i.e., whose semantics have not been defined are called *undefined behaviour* (UB). Since the standard does not impose any requirements on UB, compilers are permitted to generate code of their choice in presence of the same. In other words, compilers can assume the absence of UB in the target program, and are free to produce code without the checks for UB conditions. Further, they can produce more aggressive optimizations under such assumptions. For example, the C language standard states that writing to an array past its size is undefined. Hence, C compiler writers do not need to check the sanity of the array index during an array access. Moreover,

aggressive compilers may even remove a sanity check if the same has been added by the programmer in her C program.

C language contains hundreds of undefined behaviours [14]. All modern compilers like GCC, LLVM and ICC are known to extensively exploit UB while generating optimized code (we provide some evidence in this paper). Further, previous work on *optimization-unstable* code detection [26] reported that 40% of the 8575 C/C++ Debian Wheezy packages they tested, contain unstable code: unstable code refers to code that may get discarded during optimization due to the presence of UB. Undefined behaviour is clearly widespread. The need for UB has also been widely debated. On one hand, many textbook optimizations rely on UB semantics. For example, consider a simple `for` loop in C: `for (int i=0; i<=n; ++i)`. Now if `n` equals `INT_MAX`, then this loop would never terminate, and it would be possible for `i` to be negative inside the loop body (because `i` would wrap around after `INT_MAX`). However, several optimizations would like to depend on the loop termination property, and the loop invariant that `i >= 0` inside the loop body. Fortunately, these invariants are valid, because signed integer overflow is undefined in C (thus yielding the assumption that `++i` can never wrap around, indirectly implying that it is illegal for `n` to be equal to `INT_MAX`). On the other hand, programmers are often annoyed by these “counter-intuitive” optimizations, and some of them go to the extent of disabling certain types of UB through flags provided by the compiler. For example, the Linux kernel build process disables signed integer overflow and type based strict aliasing UB assumptions in GCC [23, 24].

Undefined behaviour semantics and their exploitation by compilers for optimization means that the compiler verification tools (e.g., translation validation) must model these semantics for more precise results. Similarly, synthesis tools and superoptimizers (e.g., [2]) must model such semantics, while comparing equivalence of the target program with the candidate synthesized program, for better optimization opportunity. An equivalence checking algorithm results in a *false negative*, i.e., incorrect equivalence failure if it does not model the UB. Previous work on simulation-based equivalence checking across compiler optimizations has primarily been done in the context of translation validation [11, 17, 18, 21, 25, 27] across selected compiler optimizations, disabling the ones that exploit language level UB. This prior work yields poor results when equivalence checks are performed across the optimizations that exploit UB. This paper addresses this issue and makes the following contributions:

- We extend the simulation relation by adding *assumptions* at each row of the simulation relation table, to model language level UB semantics. Equivalence is now computed under these assumptions, i.e., the original program and the transformed program need to be equivalent only if the corresponding assumptions are *true*. If the assumptions are *false*, the programs are still considered equivalent even if their implementations diverge. We call this the *extended simulation relation*.

```

int A[256];
int sum1 = 0; long* sum2;
void sum(int n) {
    int* p = A;
    for(int i=1; i<n+1; ++i) {
        sum1 = sum1 + *p;
        *sum2 = *sum2 + *p;
        p++;
    }
}

```

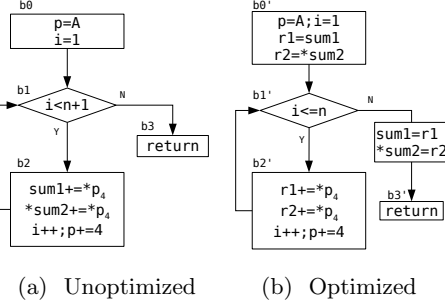


Fig. 1: An example function. `sum2` is allocated by the caller.

Fig. 2: Unoptimized and optimized, abstracted versions of the program in Fig. 1.

- We discuss the assumptions produced by different types of UB semantics and experimentally determine the types of UB that are most consequential to compiler-based optimization.
- To model aliasing based UB, which we find is heavily exploited by compilers for optimization, we present an algorithm to compute aliasing information at the IR/assembly level. Computation of aliasing information at the assembly level is necessary because the programs emitted by the compilers are in assembly. The aliasing information computed through this algorithm is used for generating UB assumptions for the extended simulation relation.

We test our ideas by comparing equivalence across unoptimized and optimized implementations of programs derived from the SPEC CPU Integer benchmark suite. The equivalence tests are performed at function granularity, i.e., an unoptimized implementation of a C function (treated as the program specification) is compared against an optimized implementation of a C function. The optimized implementations are generated using GCC and LLVM with `-O2` flag. The optimizations enabled by `-O2`, are commonly enabled by almost all software. Our overall success rate for equivalence checking across these optimizations is 81%, i.e., we successfully generate an equivalence proof, in the form of a provable simulation relation, for 81% of the equivalence checks. The success rate drops to 50% if the UB modeling is removed. Our results emphatically confirm the importance of modeling UB for checking equivalence for validation and synthesis of compiler optimizations.

2 Motivating example

Fig. 1 shows a C program which computes the sum of the first `n` elements of a global array `A` and stores the result in a global variable `sum1` and at an address `sum2`. We have deliberately used two different types of accumulators (`sum1` and `*sum2`) and `i<n+1` in the `for` loop, to demonstrate three different types of C undefined behaviour in the same example. Fig. 2a, 2b show the abstracted

unoptimized and optimized versions of the same program compiled by `gcc -O0` and `-O2` respectively. The original programs are in x86 assembly, and many other optimizations are present in the optimized version; for exposition and brevity, we have abstracted them into a C like syntax and only the UB related optimizations are shown.

The first optimization we discuss through this example, is a peephole optimization involving substitution of the check $i < n+1$ by a faster check $i \leq n$, avoiding the need to compute $n+1$. However, as such, the substitution may not seem correct because the two programs are not equivalent when $n = \text{INT_MAX}$. For $n = \text{INT_MAX}$, the loop of unoptimized program takes zero iterations ($\text{INT_MAX}+1$ wraps around to a negative number INT_MIN), while that of the optimized program loops forever (because i will always be $\leq \text{INT_MAX}$). Interestingly however, it is legal and common for C compilers to perform this optimization. This transformation is legal due to the *signed integer overflow* (SIO) assumption, that forms a part of the C undefined behaviour semantics. As per this assumption, signed integer arithmetic *shall not*¹ overflow (i.e., it is an illegal program if it causes signed integer arithmetic to overflow), and hence, the compiler need not worry about the case when overflow takes place.

The second interesting optimization in this example is the register allocation of `sum1` and `*sum2` to registers `r1` and `r2` respectively, throughout the execution of the loop. These registers containing the accumulated sum values, are written back to their respective memory locations at loop exit. Again, as such, these transformations may not seem correct: it is possible for the pointer `p`, which can belong to $[A, A+4*n)$ to alias with either (or both) of `&sum1` and `sum2`, in which case, the values stored at `p` may get modified as the loop executes, making register allocation of `sum1` and `*sum2` incorrect. It is however legal (and common) for C compilers to perform such register allocations. This is due to UB related to the following aliasing assumptions: 1) *Type based strict aliasing assumptions* (TBSA): Pointers of different types (e.g., `long*` and `int*`) shall not alias with each other (with the exception of `char*`). 2) *Out-of-bounds variable access assumptions* (OBVA): A program shall not access a memory location beyond the region of an object (variable). In our example, the TBSA assumptions guarantee that `sum2` (of type `long*`) and `p` (of type `int*`) cannot alias. Similarly, `sum2` cannot alias with `&sum1` (of type `int*`). Further, the OBVA assumptions guarantee that `p` cannot point beyond the object `A`, i.e., `p` must belong to $[A, A+4*256)$. This implies that `p` cannot alias with `&sum1`, as `sum1` and `A` are distinct regions. With these assumptions, it is indeed legal to register-allocate `sum1` and `*sum2` throughout the loop execution.

The programs in Fig. 2a, 2b can be shown to be equivalent only if the UB assumptions are modeled and used in the simulation-based proof. In this paper, we contribute algorithms to model and use these UB assumptions in a simulation-based proof, and show their effectiveness for computing equivalence across compiler transformations on a general purpose code. Sec. 3 discusses the notion of the extended simulation relation that uses undefined behaviour as-

¹ Phrasing is taken from the C standard.

sumptions to correctly decide equivalence in the presence of UB. Sec. 4 discusses algorithms to generate these UB assumptions, for use in the extended simulation relation.

3 Extended simulation relation (with assumptions)

A simulation relation [17, 18] between two programs can be used to establish equivalence across the two programs. It has been used extensively in previous work on equivalence checking and translation validation [11, 17, 18, 20, 27]. A simulation relation is a witness of the equivalence between two programs. Given a valid simulation relation, proving equivalence is straight-forward; however the construction of a simulation relation is undecidable in general. We leverage previous work on automatic construction of a simulation relation across two programs, where the second program is the compiler-optimized version of the first program. In addition, we extend previous work to model and use UB assumptions, to allow equivalence computation in the presence of UB semantics. Equivalence is now conditional on these assumptions, i.e., the equivalence proof may fail if these assumptions are discounted.

The relevant assumptions are computed at each program location of the unoptimized program specification. These assumptions are based on a best-effort static analysis of the program: for example, if the program involves arithmetic on a signed integer variable, then the corresponding SIO assumption is inferred at that program location. Some assumptions can be inferred directly from program syntax, while others may require a deeper static analysis. In general, the sophistication of the static analysis required to infer the undefined behaviour assumptions, ought to match the sophistication of the analyses used by the optimizer. SIO and TBSA assumptions are examples of assumptions that can be inferred through straight-forward syntactic analysis of the program, while the OBVA assumptions usually require a deeper alias analysis, the kind used by modern compilers for optimization. We discuss this latter analysis in Sec. 4. In this section, we assume that such assumptions are already available at the respective program locations, and we discuss their effect on the required simulation relation.

Let $Prog_A$ be the unoptimized program specification and $Prog_B$ be the optimized implementation. $Prog_A$ specification also includes a map from the program locations to the corresponding UB assumptions ($Assum$). An extended simulation relation is represented as a table, where each row is a tuple $((L_A, L_B), Assum[L_A], P)$ such that L_A and L_B are program locations in $Prog_A$ and $Prog_B$ respectively, $Assum[L_A]$ is the set of assumptions in $Prog_A$ at location L_A , and P is a set of invariants on the live program variables at locations L_A and L_B . A tuple $((L_A, L_B), Assum[L_A], P)$ represents that the invariants P hold whenever the two programs are at L_A and L_B respectively, *assuming* all the UB assumptions at *all* $Prog_A$ program locations ($Assum$) hold.

An extended simulation relation is valid if the invariants at each location pair are inductively provable from invariants and UB assumptions at the predecessor

Location	Assumption	Invariants (P)
(b0,b0')	True	$n_A = n_B, A_A = A_B, \&sum1_A = \&sum1_B, sum2_A = sum2_B, M_A =_{\Delta} M_B$
(b1,b1')	$(n_A \neq INT_MAX) \wedge (\&sum1_A \neq p_A) \wedge (sum2_A \neq p_A) \wedge (sum2_A \neq \&sum1_A)$	$sl_4(M_A, \&sum1_A) = r1_B,$ $sl_4(M_A, sum2_A) = r2_B, n_A = n_B, i_A = i_B,$ $A_A = A_B, p_A = p_B, \&sum1_A = \&sum1_B,$ $sum2_A = sum2_B, M_A =_{\Delta \cup \{\&sum1_A, sum2_A\}} M_B$
(b3,b3')	True	$M_A =_{\Delta} M_B$

Init: $n_A = n_B, A_A = A_B, \&sum1_A = \&sum1_B, sum2_A = sum2_B, M_A =_{\Delta} M_B$

Fig. 3: Extended simulation relation for the programs in Fig. 2. (b0, b0') and (b3, b3') are the entry and exit rows respectively. A_A and $\&sum1_A$ are the base addresses of the globals A and sum1 respectively in $Prog_A$. $sl_4(M, addr)$ represents 4 bytes of data read in memory (M) at address $addr$. $=_{\Delta}$ represents equivalent memory states except at Δ ; Δ represents the stack region. Init represents equivalence of inputs.

location pairs. Notice that the UB assumptions do not need to be proven. Invariants at the entry location (pair of entry locations of the two programs) represent the equivalence of program inputs ($Init$); the base case of this inductive proof. Finally, if we can thus inductively prove equivalence of the return values at exit location (pair of exits of the two programs), we have established the programs to be equivalent. For C functions, the return values include the state of the heap and global variables. Formally, an extended simulation relation is valid if:

$$Init \Leftrightarrow invariants_{(Entry_A, Entry_B)}$$

$$\forall_{(L'_A, L'_B) \rightarrow (L_A, L_B)} Assum[L'_A] \wedge invariants_{(L'_A, L'_B)} \Rightarrow_{(L'_A, L'_B) \rightarrow (L_A, L_B)} invariants_{(L_A, L_B)}$$

Here $invariants_{(L_A, L_B)}$ represents the conjunction of invariants in the extended simulation relation for the location pair (L_A, L_B) , $Init$ is the input equivalence condition at the entry of the two programs, L'_A and L'_B are predecessors of L_A and L_B in programs $Prog_A$ and $Prog_B$ respectively, and $\Rightarrow_{(L'_A, L'_B) \rightarrow (L_A, L_B)}$ represents implication over the paths $L'_A \rightarrow L_A$ and $L'_B \rightarrow L_B$ in programs $Prog_A$ and $Prog_B$ respectively.

Fig. 3 shows an extended simulation relation which establishes the equivalence across the programs in Fig. 2a and 2b. The exit row of this extended simulation relation denotes equivalence of memory states (modulo stack and local variables) at exit, representing the equivalence of globals variables $\{sum1, A\}$ and values at pointer sum2 and the remaining unused heap. This simulation relation is only provable when the UB assumptions are used in the inductive proof. For example, without the assumptions, the invariant $sl_4(M_A, \&sum1_A) = r1_B$ of the second row is not provable on edge $(b1, b1') \rightarrow (b1, b1')$ (sl_4 represents the memory-read of four bytes; see Fig. 3 caption).

Type of undefined behaviour	Description
Signed integer overflow (SIO)	Signed integer arithmetic cannot overflow
Type based strict aliasing (TBSA)	Pointers of different types cannot alias (barring exceptions like <code>char *</code>)
Dereferenced addresses not null	An address that has been dereference cannot be zero
Shift operand bounds	If a value X is shifted left/right by another value S , then $S \geq 0$ and $S < \text{numbits}(X)$ ($\text{numbits}(X)$ is the number of bits used to represent X)
Type alignment	A value X of type T must be aligned to the size of T
No divide by zero	The divisor of a division operation cannot be zero

Table 1: Examples of types of C undefined behaviour that can be modeled through syntactic analysis of the program.

4 Modeling undefined behaviour assumptions

We now discuss how to obtain the UB assumptions for the simulation relation. We first generate these assumptions on the unoptimized program specification Prog_A , for each location, through static analysis of the program. At the time of the construction of the simulation relation, for every row (L_A, L_B) , the assumptions corresponding to L_A are added in the simulation relation. In other words, the UB assumptions are inferred for the unoptimized program, and used during the construction and proof of the simulation relation.

The algorithm to infer the UB assumptions, depends on the type of the UB. For example, the assumptions for many types of UB can be inferred purely syntactically — see Table 1 for some examples. Such syntactic analysis and modeling of UB has also been used previously for the verification of manually written peephole optimizations in LLVM [16].

The OBVA undefined behaviour assumptions are an example of UB that require a relatively deeper static alias analysis. This is because the production quality compilers typically implement a similar alias analysis for better optimization opportunity. The static alias analysis provides a *may-alias* relation between program pointers and program *variables*. The program variables include all the global and local variables defined by the programmer. Further, to model aliasing in heap and stack, we include two special “variables”, called “stack” and “heap”. Thus, a pointer value in the program may alias with one or more of the user-defined variables, and/or with the stack/heap². Based on this analysis, we infer assumptions indicating that a program pointer must point within the memory regions belonging to the variables with which it may alias:

$$\text{aliasing_assumptions}_p \Leftrightarrow \bigvee_{v \in \{u: \text{may_alias}(p, u)\}} (p \geq v_{\text{begin}} \wedge p < v_{\text{end}})$$

² While a stack is not a part of the program’s language level semantics, it gets introduced by the compiler in the assembly implementation.

Here p represents a pointer value, v is a program variable p , and $[v_{begin}, v_{end})$ represents the region of memory occupied by variable v . Further, invariants encoding the mutual-disjointness of regions associated with each program variable, and for the stack and heap, are added through conditions on the respective v_{begin} and v_{end} values.

In our running example of Fig. 1, the alias analysis infers that p may alias with only the array variable A . Further, because A and $sum1$ are different variables, their memory regions are mutually disjoint, thus implying that p cannot alias with $sum1$.

This alias analysis, to infer the variables with which a program pointer may alias, is similar to the previous work on alias analysis for assembly code [4]. The alias analysis need not be precise, but needs to be sound, i.e., the may-alias relation for a pointer p must include all variables that a pointer may actually alias with (over-approximation). We next describe the two analyses used by us to infer the may-alias relation.

4.1 May-alias analysis

To compute the may-alias relation, we first compute two relations, *linearly-related* (lr) and *may-depend-on* (dep) between program pointers and program variables (including stack and heap). The lr relation indicates the variable with which a program pointer is linearly-related, i.e., *based-on*. In other words, if a program pointer is at an offset from the address of a program variable then it is lr with that program variable. For example, a pointer $p=v+10$ or $p=v+i$ (for some arbitrary variable i) are both lr with the variable address v . On the other hand, $p=*v$ is *not* lr with v (even though p *may depend on* v , as we discuss later). In our running example of Fig. 1, p is lr with A . The C type system guarantees that a pointer may be lr with at most one program variable³. Also, if a program pointer p is lr with a program variable A , then p may alias with A , and *cannot* alias with any other variable (including stack/heap). A pointer can at most be lr with one variable.

In addition to the lr relation, we compute another relation called “may-depend-on” dep . This relation indicates the variables on which a program pointer may depend on, i.e., the variables whose address may potentially influence the value of this program pointer. If the address of a variable may not influence the value of a pointer, then that pointer may be assumed to not alias with the aforementioned variable. Note that $lr(p, v)$ implies $dep(p, v)$.

The may-alias relation between a pointer p and program variable v is computed in terms of the linearly-related and may-depend-on relations as follows:

$$may_alias(p, v) \Leftrightarrow dep(p, v) \wedge \bigwedge_{w \in (V-v)} (\neg lr(p, w))$$

³ A violation of this type-system, through type-punning for example, falls into the realm of UB.

Here V is the set of all program variables. In other words, we assume that a pointer p may alias with a variable v if it may depend on v , *and* it is not linearly-related to any other variable $w \neq v$ in V ⁴.

4.2 Computing linearly-related and may-depend-on relations

Computing both lr and dep relations involves a forward dataflow analysis on the program’s control flow graph. These relations are initialized at program entry with conservative assumptions, and they are computed at each intermediate program location by analyzing transfer functions of the incoming control-flow edges. In our setting, each program represents a C function body, and the calling conventions of the compiler are used to initialize the relations at the entry node, i.e., we assume that the function arguments *may depend on* any of the global variables and/or the heap, but are independent of the stack and local variables of the function. Further, we assume that the function arguments are *not* lr with any global variable. Together, these assumptions at program entry specify that the function arguments may alias with all the program’s global variables and the heap, but cannot alias with the function’s stack/local variables.

The lr analysis across a control-flow edge involves a simple syntactic analysis of the expression trees of the transfer function on that edge. This syntactic analysis involves inference rules of the type: $lr(p, v) \Rightarrow lr(p \oplus X, v)$. i.e., if p is known to be lr with v , then $p \oplus X$ (for any expression X that may potentially depend on other variables $w \neq v$) is also lr with v . “ \oplus ” represents the addition and subtraction operators; we further generalize these rules to operations involving bitwise masking of lower-order bits of a pointer (a common operation in compiled code). If these inference rules cannot decide a pointer p to be lr with a variable v , then we conservatively assume that p is *not* lr with v (over-approximation). At all internal nodes (except the start node), we initially assume all pointers to be lr with all variables (\top), and refine the relations iteratively till a fixed point is reached. As discussed earlier, at the start node, we assume that none of the function arguments are lr with any of the variables. This information on lr relations flows from the program entry to all intermediate program locations, through transfer functions. The meet operator for this lr dataflow analysis is *intersection*, i.e., a pointer is lr with a variable only if it is lr on *all* possible program paths.

Similarly, the dep analysis across a control-flow edge also involves a syntactic analysis on the expression trees of the corresponding transfer function. The syntactic analysis involves inference rules of the type: $dep(p, v) \Rightarrow dep(OP(\dots, p, \dots), v)$. i.e., if p may depend on v , then any value derived from p (through any operation OP that uses p as an argument) may also depend on v . At the entry node, we conservatively assume that the function arguments may depend on any of the global variables or on the heap. At all intermediate nodes, we initialize by assuming that the pointers do not depend on any of the variables (\top). At

⁴ As discussed earlier, the C type system ensures that if p is linearly-related to a variable w , then p cannot alias with any other variable $v \neq w$.

each iteration, we refine this may-depend-on relation at every node by analyzing the expression trees of the transfer function of each incoming edge. The meet operator for the *dep* relation is *union*, i.e., a pointer may depend on a variable if it depends on that variable on *any* program path.

Unlike compilers, our alias analysis needs to work for assembly code where pointer arithmetic is much more common. The *lr* relation is intended to capture such pointer arithmetic. Also, the modeling of stack is unique to assembly code. Our algorithm, which over-approximately computes the may-alias relation through *lr* and *dep* relations, is sound and efficient (polynomial in the size of the program and quite fast in practice), and captures the common patterns in compiled code. A more expensive analysis can potentially yield more precise may-alias relations.

5 Inferring the simulation relation

Automatic construction of the simulation relation has been well studied in prior work [5, 11, 17, 18, 20, 27]. Much previous work attempts to first discover a correspondence between program locations across the two programs (correlation (L_A, L_B)) in a first pass, and then attempts to find invariants (P) over the locations in a best-effort second pass. In contrast, our algorithm searches for the correlation simultaneously with the search for the invariants, resulting in a more flexible and robust system. We succinctly outline here, our correlation algorithm to automatically construct a provable simulation; a more detailed discussion is available in [3].

Our algorithm incrementally constructs a *joint transfer function graph* (JTFG) representing the partial simulation relation computed so far. A JTFG is a graph with nodes and edges. A JTFG node (L_A, L_B) represents a pair of program nodes L_A and L_B (indicating that $Prog_A$ is at L_A and $Prog_B$ is at L_B). Similarly, a JTFG edge $(L'_A, L'_B) \rightarrow (L_A, L_B)$, represents a pair of transitions $L'_A \rightarrow L_A$ and $L'_B \rightarrow L_B$ in $Prog_A$ and $Prog_B$ respectively. Thus, a transition across a JTFG edge encodes transitions in the two programs respectively. Each JTFG node (L_A, L_B) contains invariants relating the live variables at locations L_A and L_B in the two programs respectively. To model UB, the JTFG nodes further encode the UB assumptions. Recall that these assumptions have already been computed through static analysis for locations in $Prog_A$; the assumptions at location L_A in $Prog_A$ appear in all JTFG nodes containing L_A . Further, for each JTFG edge, *edge conditions* (*edgecond*) of its two individual constituent program control-flow edges (belonging to $Prog_A$ and $Prog_B$ resp.) should be equivalent. An edge condition represents the condition under which that edge is taken, as a function of the live variables at the source location of that edge.

The algorithm for constructing a JTFG is presented in Algorithm 1. The JTFG is initialized with a single node, representing the pair of entry locations of the two programs. The `CorrelateEdges()` function picks one $Prog_B$ edge, say $edge_B$, at a time and tries to identify paths in the unoptimized program ($Prog_A$) that have an equivalent *path condition* to $edge_B$'s edge condition. Several candi-

```

Function CorrelateEdges(jtfg, edgesB)
  if edgesB is empty then
    | return LiveValuesAtExitAreEquivalent(jtfg)
  end
  edgeB ← RemoveFirst(edgesB)
  edgesA ← GetEdgesTillUnroll(ProgA, edgeB, μ)
  foreach edgeA in edgesA do
    | jtfg' = AddEdge(jtfg, edgeA, edgeB)
    | PredicatesGuessAndCheck(jtfg')
    | if IsEqualEdgeConditions(jtfg') ∧ CorrelateEdges(jtfg', edgesB) then
      | | return true
    | end
  end
  return false

```

Algorithm 1: Algorithm to construct the JTFG (simulation relation). $edges_B$ is a list of edges in $Prog_B$ in depth-first search order. The $AddEdge()$ function returns a new JTFG $jtfg'$, formed by adding the edge to the old JTFG $jtfg$.

date paths are attempted up to an unroll factor μ ($GetEdgesTillUnroll()$). All candidate paths must originate from a $Prog_A$ location that has already been correlated with the source location of $edge_B$. The path condition of a path is formed by appropriately composing the edge conditions of the edges belonging to that path. The edge $edge_B$ is chosen in depth-first search order from $Prog_B$, and also dictates the order of incremental construction of the JTFG. The equivalence of the edge condition of $Prog_B$ with the path condition of $Prog_A$ is computed based on the invariants inferred so far at the already correlated JTFG nodes ($IsEqualEdgeConditions()$). These invariants, inferred at each step of the algorithm, are computed through a Houdini-style [7] guess-and-check procedure. The guesses are synthesized from a grammar, through syntax-guided synthesis of invariants [1] ($PredicatesGuessAndCheck$). The unroll factor μ allows equivalence computation across transformations involving loop unrolling.

These correlations for each edge ($edge_B$) are determined recursively to allow backtracking (see the recursive call to $CorrelateEdges()$). If at any stage, an edge ($edge_B$) cannot be correlated with a path in $Prog_A$, the function returns with a failure, prompting the caller frame in this recursion stack, to try another correlation for a previously correlated edge. In theory, this backtracking can be exponential in the number of edges, but in practice, backtracking is rare, especially because we prioritize the candidate source paths for correlation, in increasing order of their unrolling factor. Because most compiler transformations do not involve unrolling, backtracking is rare in this scenario.

$PredicatesGuessAndCheck()$ synthesizes invariants through the following grammar of guessing: $G = \{ \star_A \oplus \star_B, M_A =_{\star_A \cup \star_B} M_B \}$, where operator $\oplus \in \{<, >, =, \leq, \geq\}$ and \star_A and \star_B represent the program values (represented as symbolic expressions) appearing in $Prog_A$ and $Prog_B$ respectively. The guesses are formed through a Cartesian product of values in $Prog_A$ and $Prog_B$ using

the patterns in \mathbb{G} . Our checking procedure is a fixed point computation which keeps eliminating the unprovable predicates, until only provable predicates remain (similar to Houdini). At each step, for each guessed predicate at each node, we try to prove it from every predecessor node using the current invariants and assumptions at the predecessor node (as also described in Sec. 3).

For our running example in Fig. 2a, 2b, the JTFG nodes and edges determined through our algorithm are $\{(b0, b0'), (b1, b1'), (b3, b3')\}$ and $\{(b0, b0') \rightarrow (b1, b1'), (b1, b1') \rightarrow (b1, b1'), (b1, b1') \rightarrow (b3, b3')\}$ respectively. Further, the algorithm is able to infer the required invariants (shown in the last column of Fig. 3) to complete the equivalence proof.

6 Implementation and Experiments

To demonstrate the impact of undefined behaviour assumptions on compiler optimization, we compute equivalence of C functions across unoptimized (-O0) and optimized (-O2) x86 binaries produced by compiling C programs through production compilers, GCC and LLVM with and without UB models. We disable function inlining during compilation, as our prototype implementation cannot reason about inter-procedural optimizations. Even after disabling inlining, the average speedup across the compiler optimizations on these programs is 1.72x over clang-O0. To be able to reconstruct the C-level information, required for modeling UB and equivalence checking, we enable a few additional flags during the compilation (namely `-g` and `-reloc`) to generate debug information and relocation headers respectively. We assume that the binaries contain the symbol table (i.e., are unstripped), which along with relocation headers allow accurate renaming of memory addresses to global variable symbols. Further the debug information provides the signature and types of the variables and functions. Both GCC and LLVM support these compile-time options, and these options have no impact on the runtime of the executable.

The functions are drawn from four SPEC benchmarks: `bzip2` (compression utility), `gzip` (compression utility), `mcf` (combinatorial optimization) and `parser` (word processing). The number of global variables in these benchmarks is 100, 212, 43 and 223 respectively. We compiled each program with both compilers to produce 16 binaries (8 unoptimized and 8 optimized), representing a total of 1058 pairs of unoptimized and optimized assembly functions (ignoring the identical `glibc` functions). Among these pairs, 714 functions had at least one loop in them (cyclic functions). The average number of assembly LOC and C-LOC for these functions is 112 and 35 respectively. We ignored the functions containing floating point operations (14 functions), as our semantic model for x86 floating point instructions is incomplete.

We performed experiments to demonstrate the significance of the three types of UB discussed in Sec 2, namely SIO, TBSA, and OBVA assumptions. We estimate the presence of UB based optimizations for each benchmark and compiler option, by performing the equivalence check twice, for each function, with and without using the UB assumption. If an equivalence check for a function pair

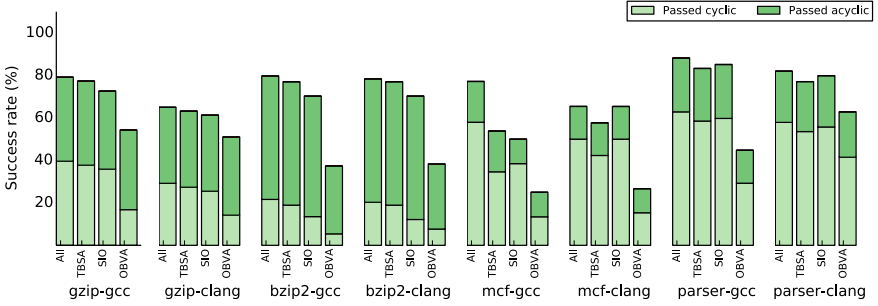


Fig. 4: For every benchmark-compiler option, the first bar shows the success rates when we model all three UB. The remaining three bars show the success rates when a particular type of UB among three (TBSA, SIO, OBVA) is not modeled. Each bar individually shows the contribution to the success rates by cyclic and acyclic functions.

passes with the UB assumption but fails without the assumption, then we assume that the compiler has exploited the respective undefined behaviour towards optimizing the function. The plot in Fig. 4 shows the success rates for each compiler and each benchmark for four different cases: the first bar represents the success rate when all three undefined behaviours are modeled; the second, third and fourth bars represent the cases when TBSA, SIO and OBVA assumptions are not modeled respectively. For SIO and TBSA, we employ the compiler flags `fno-strict-overflow` and `fno-strict-aliasing` to differentially estimate the impact of these assumptions. These flags enable/disable the SIO and TBSA assumptions while performing optimizations. If our equivalence check passes when these assumptions are disabled by the compiler, but fails when these assumptions are enabled by the compiler, we assume that the compiler is leveraging these assumptions for optimization. For OBVA, we simply turn on/off our alias analysis (Sec. 4) to determine the effect of OBVA assumptions.

The overall average success rates for equivalence checking across the four cases are 81%, 76%, 77% and 50%. As expected, the success rates are lower when a certain type of UB is not modeled. The drop in success rates, when a UB is not modeled with respect to the first bar (where all three types of UB are modeled), indicates the impact of the respective type of UB on compiler optimization. The drop in success rates due to non-modeling of OBVA assumptions is 31 percentage points. In contrast, the drop due to non-modeling of SIO and TBSA assumptions is only 4 and 5 percentage points respectively. These experiments confirm (a) the widespread impact of undefined behaviours on compiler optimizations and (b) throw light on the relative impact of different types of C undefined behaviour on optimization.

Our experiments also led to the discovery of a bug in GCC-4.1.0 related to the semantics of `fno-strict-aliasing` [8]. This flag is used to disable the optimizations related to TBSA. However, for certain functions, GCC-4.1.0 was using TBSA assumptions even while compiling with this flag.

7 Related Work

Modeling of UB for verification has previously been studied in Alive [16], where *acyclic* peephole optimization patterns of the `InstCombine` pass in LLVM are verified. These optimizations could potentially involve UB assumptions, and hence modeling of UB becomes necessary. The typical verification target for Alive is a few lines of optimization pattern representing a single optimization. In contrast, our verification targets involve concrete programs (with up to 1000s of lines) and containing *multiple* composed compiler optimizations. Alive models UB involving undefined values, poison values and instruction attributes like `nsw` (signed integer overflow), the kind that can be modeled through a simple syntactic analysis of the LLVM peephole optimization pattern. For example, the presence of UB attributes like `nsw`, `undef`, etc., in the optimization pattern directly indicates the UB assumptions. Aliasing based UB involving OBVA requires an alias analysis, and Alive did not consider this in their work. Our work is directed towards studying the common transformations in end-to-end compiler optimization, and we find that UB involving OBVA is the most commonly exploited for optimization in both GCC and LLVM. We believe that our alias analysis can also benefit Alive interested in capturing aliasing based UB assumptions. Another major difference between Alive and our work is that Alive verifies acyclic optimization patterns, while we generalize the ideas to simulation-based equivalence across programs containing loops.

Our work overlaps with previous work on detection of *unstable* code, STACK [26]. STACK classifies unstable code as the code whose semantics are sensitive to UB. The underlying assumption of this work is that if an optimizer discards/modifies the (unstable) code due to the presence of UB, the resulting logic may behave differently from what the programmer intended. While STACK identifies certain important types of unstable code through static pattern-matching on LLVM IR, it also leaves out many. Aliasing based UB stands out as an example of UB not considered by STACK. It should be straight-forward to extend STACK by employing an alias analysis similar to our work. Our simulation-based equivalence proof construction approach is in contrast with the largely syntactic pattern matching approach adopted by STACK. It would be instructive to study the merits of applying a semantic procedure like ours, to the detection of unstable code.

Our *lr* and *dep* analyses, resemble previous work on alias analysis for executable code by Debray et. al. [4]. The authors of this work noted that alias analysis for executable code requires reasoning about pointer arithmetic, and hence proposed special modeling for the `add` and `mult` opcodes, as these were the most commonly encountered opcodes for pointer manipulation on the RISC architecture they considered. However, because their analysis is syntactic in nature, it introduces imprecisions in common situations involving store and subsequent load of a pointer to/from memory. In such situations where a syntactic analysis does not provide enough information, the alias information would be conservatively widened to \perp in their approach. Their empirical evaluations reflect these imprecisions. Our approach works on de-sugared expressions obtained

from machine opcodes, involving standard bitvector and boolean operators. Also, our memory model allows reasoning about stores followed by loads to identical locations (without other intervening conflicting stores), thus capturing the common pattern of pointers getting saved to stack slots for future reference. This semantic treatment lends robustness to our analysis, and makes it independent of the underlying machine ISA. In another related work on alias analysis, Fernandez and Espasa [6] attempted to remove the imprecisions discussed in [4], by sacrificing soundness guarantees. Sacrificing soundness is not acceptable in our setting. The authors of both these previous works on alias analysis for executable code were interested in link-time optimizations; unlike us, they do not describe a model for reasoning about UB using this obtained aliasing information.

Translation validation infrastructure (TVI) [17] verified five IR passes of compilation of gcc-2.91 and Linux-2.2 by GCC. The passes verified were branch optimization, common-subexpression elimination (CSE), loop unrolling and inversion, register allocation, and instruction scheduling. Similarly, value-graph translation validation for LLVM has been performed in at least two independent efforts [21, 25], albeit only across a known set of nine selected transformations, namely, dead-code elimination, global value numbering, constant propagation, loop-invariant code motion, loop deletion, loop unswitching, dead-store elimination, partial-redundancy elimination, and basic block placement. Neither of these approaches model UB, or study their significance on compiler optimization. Overall, our success rates for equivalence checking are comparable (and often better) to all these previous efforts, albeit in a much more generalized setting (with almost no assumptions on the transformations that are enabled). To our knowledge, our experiments are the first to demonstrate the significance of UB on compiler optimization.

There are more approaches to translation validation and equivalence checking with different settings and goals (e.g., [5, 9, 10, 12, 13, 15, 19, 20, 22, 27, 28]). All previous simulation-based equivalence checkers can also be extended with UB assumptions, to capture a larger set of compiler transformations.

There are hundreds of types of UB in C, and some of them have been bitterly debated in the past [23, 24]. We believe that this approach to quantifying the impact of different types of UB on compiler optimization, can bring some insight and basis for such debates. For example, our limited investigations in this work indicate the overwhelming relative significance of out-of-bounds variable access assumptions (for optimization), compared to other types of UB like signed integer overflow and type based strict aliasing assumptions. We hope that this work triggers more such studies across a wider variety of UB in future.

References

1. Alur, R., Bodik, R., Juniwal, G., Martin, M.M.K., Raghothaman, M., Seshia, S., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: FMCAD, 2013
2. Bansal, S., Aiken, A.: Automatic generation of peephole superoptimizers. ASPLOS XII (2006)

3. Dahiya, M., Bansal, S.: Black-box equivalence checking across compiler optimizations. In: APLAS '17 (2017)
4. Debray, S., Muth, R., Weippert, M.: Alias analysis of executable code. POPL '98
5. Felsing, D., Grebing, S., Klebanov, V., Rümmer, P., Ulbrich, M.: Automating regression verification. ASE '14 (2014)
6. Fernández, M., Espasa, R.: Speculative alias analysis for executable code. PACT '02
7. Flanagan, C., Leino, K.: Houdini, an annotation assistant for esc/java. In: FME 2001: Formal Methods for Increasing Software Productivity. LNCS (2001)
8. GCC Bugzilla - Bug 68480, https://gcc.gnu.org/bugzilla/show_bug.cgi?id=68480
9. 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. ESEC/FSE 2013
10. Kanade, A., Sanyal, A., Khedker, U.P.: Validation of gcc optimizers through trace generation. Softw. Pract. Exper. (2009)
11. Kundu, S., Tatlock, Z., Lerner, S.: Proving optimizations correct using parameterized program equivalence. PLDI '09 (2009)
12. Lahiri, S., Hawblitzel, C., Kawaguchi, M., Rebelo, H.: Symdiff: A language-agnostic semantic diff tool for imperative programs. In: CAV '12 (2012)
13. Lahiri, S., Sinha, R., Hawblitzel, C.: Automatic rootcausing for program equivalence failures in binaries. In: Computer Aided Verification (CAV'15) (2015)
14. Lee, J., Kim, Y., Song, Y., Hur, C.K., Das, S., Majnemer, D., Regehr, J., Lopes, N.P.: Taming undefined behavior in llvm. PLDI 2017 (2017)
15. Leung, A., Bounov, D., Lerner, S.: C-to-verilog translation validation. In: MEM-OCODE (2015)
16. Lopes, N.P., Menendez, D., Nagarakatte, S., Regehr, J.: Provably correct peephole optimizations with alive. PLDI 2015
17. Necula, G.C.: Translation validation for an optimizing compiler. PLDI '00 (2000)
18. Pnueli, A., Siegel, M., Singerman, E.: Translation validation. TACAS '98 (1998)
19. Poetzsch-Heffter, A., Gawkowski, M.: Towards proof generating compilers. Electron. Notes Theor. Comput. Sci. (2005)
20. Sharma, R., Schkufza, E., Churchill, B., Aiken, A.: Data-driven equivalence checking. OOPSLA '13 (2013)
21. Stepp, M., Tate, R., Lerner, S.: Equality-based translation validator for llvm. CAV'11 (2011)
22. Strichman, O., Godlin, B.: Regression verification - a practical way to verify programs. In: Verified Software: Theories, Tools, Experiments, vol. 4171 (2008)
23. Torvalds, L.: <https://lkml.org/lkml/2007/5/7/213>
24. Torvalds, L.: <https://gcc.gnu.org/ml/gcc/2002-01/msg00395.html>
25. Tristan, J.B., Govereau, P., Morrisett, G.: Evaluating value-graph translation validation for llvm. PLDI '11
26. Wang, X., Zeldovich, N., Kaashoek, M.F., Solar-Lezama, A.: Towards optimization-safe systems. SOSP '13 (2013)
27. Zaks, A., Pnueli, A.: Covac: Compiler validation by program analysis of the cross-product. FM '08 (2008)
28. Zuck, L., Pnueli, A., Fang, Y., Goldberg, B.: Voc: A methodology for the translation validation of optimizing compilers 9(3) (2003)

Hardware and Software: Verification and Testing
13th International Haifa Verification Conference, HVC
2017, Haifa, Israel, November 13-15, 2017,
Proceedings
Strichman, O.; Tzoref-Brill, R. (Eds.)
2017, XXII, 253 p. 47 illus., Softcover
ISBN: 978-3-319-70388-6